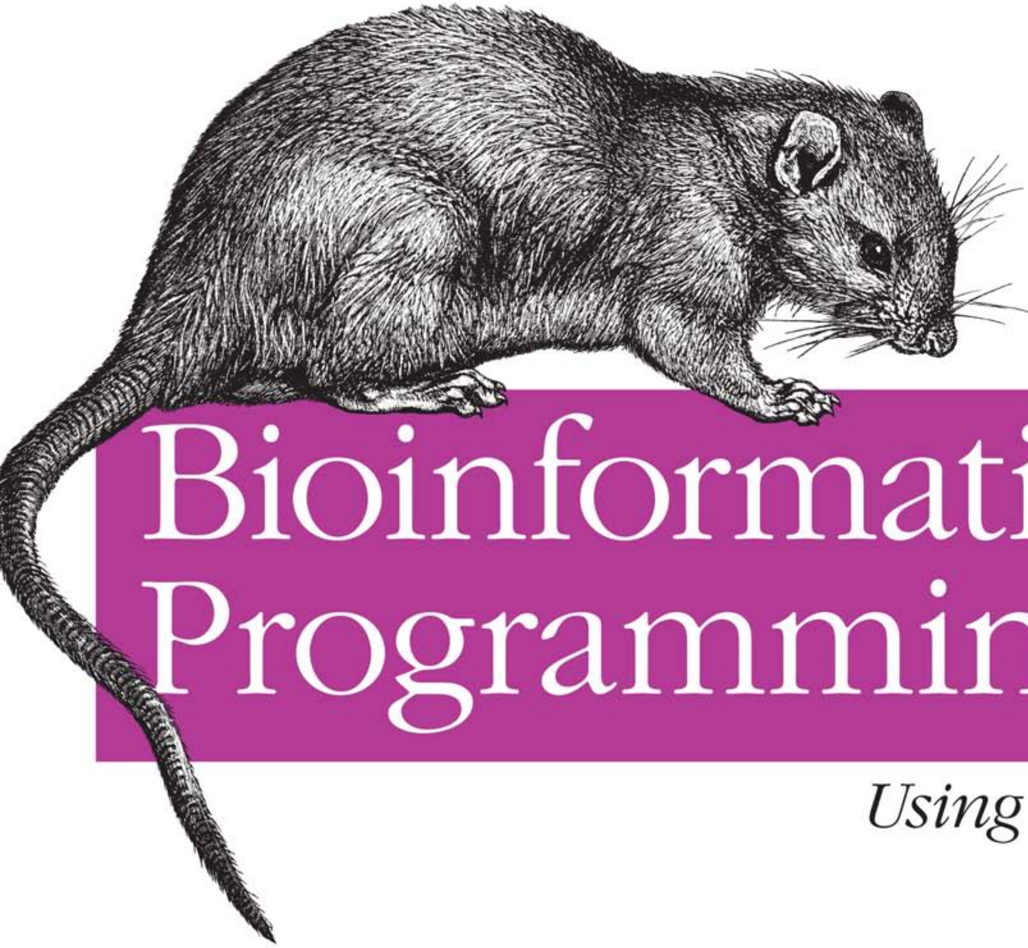
Practical Programming for Biological Data

Covers Python 3



Bioinformatics Programming

Using Python

O'REILLY®

Mitchell L Model

Bioinformatics Programming Using Python

Powerful, flexible, and easy to use, Python is an ideal language for building software tools and applications for life science research and development. This unique book shows you how to program with Python, using code examples taken directly from bioinformatics. In a short time, you'll be using sophisticated techniques and Python modules that are particularly effective for bioinformatics programming.

Bioinformatics Programming Using Python is perfect for anyone involved with bioinformatics—researchers, support staff, students, and software developers interested in writing bioinformatics applications. You'll find it useful whether you already use Python, write code in another language, or have no programming experience at all. It's an excellent self-instruction tool, as well as a handy reference when facing the challenges of real-life programming tasks.

- Become familiar with Python's fundamentals, including ways to develop simple applications
- Learn how to use Python modules for pattern matching, structured text processing, online data retrieval, and database access
- Discover generalized patterns that cover a large proportion of how Python code is used in bioinformatics
- Learn how to apply the principles and techniques of object-oriented programming
- Benefit from the "tips and traps" section in each chapter

"*Mitchell Model has done the bioinformatics community a real service. In biology, already an information-heavy science, new generations of sequencers have opened the floodgates of data. The Python programming techniques introduced here will enable you to transform that data into knowledge.*"

—Jim Tisdall
Author of Beginning Perl for Bioinformatics

Mitchell L Model has been focused on bioinformatics for 15 years and has worked with a wide range of platforms, languages, technologies, and domains. As an independent consultant, he provides training, mentoring, tools, and support to software development groups learning new technologies and practices.

O'REILLY®
oreilly.com

US $59.99 CAN $74.99
ISBN: 978-0-596-15450-9

55999
9 780596 154509

Bioinformatics Programming Using Python

Bioinformatics Programming Using Python

Mitchell L Model

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Bioinformatics Programming Using Python
by Mitchell L Model

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Sarah Schneider
Copyeditor: Rachel Head
Proofreader: Sada Preisch
Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
December 2009: First Edition.

This book uses RepKover, a durable and flexible lay-flat binding.

ISBN: 978-0-596-15450-9

[M]

1259959883

Table of Contents

Preface

This preface provides information I expect will be important for someone reading and using this book. The first part introduces the book itself. The second talks about Python. The third part contains other notes of various kinds.

Introduction

I would like to begin with some comments about this book, the field of bioinformatics, and the kinds of people I think will find it useful.

About This Book

The purpose of this book is to show the reader how to use the Python programming language to facilitate and automate the wide variety of data manipulation tasks encountered in life science research and development. It is designed to be accessible to readers with a range of interests and backgrounds, both scientific and technical. It emphasizes practical programming, using meaningful examples of useful code. In addition to meeting the needs of individual readers, it can also be used as a textbook for a one-semester upper-level undergraduate or graduate-level course.

The book differs from traditional introductory programming texts in a variety of ways. It does not attempt to detail every possible variation of the mechanisms it describes, emphasizing instead the most frequently used. It offers an introduction to Python programming that is more rapid and in some ways more superficial than what would be found in a text devoted solely to Python or introductory programming. At the same time, it includes some advanced features, techniques, and topics that are often omitted from entry-level Python books. These are included because of their wide applicability in bioinformatics programming, and they are used extensively in the book's examples.

Python's installation includes a large selection of optional components called "modules." Python books usually cover a small selection of the most generally useful modules, and perhaps some others in less detail. Having bioinformatics programming as this book's target had some interesting effects on the choice of which modules to discuss, and at what depth. The modules (or parts of modules) that are

covered in this book are the ones that are most likely to be particularly valuable in bioinformatics programming. In some cases the discussions are more substantial than would be found in a generic Python book, and many of the modules covered here appear in few other books. Chapter 6, in particular, describes a large number of narrowly focused "utility" modules.

The remaining chapters focus on particular areas of programming technology: pattern matching, processing structured text (HTML and XML), web programming (opening web pages, programming HTTP requests, interacting with web servers, etc.), relational databases (SQL), and structured graphics (Tk and SVG). They each introduce one or two modules that are essential for working with these technologies, but the chapters have a much larger scope than simply describing those modules.

Unlike many technical books, this one really should be read linearly. Even in the later chapters, which deal extensively with particular kinds of programming work, examples will often use material from an earlier chapter. In most places the text says that and provides cross-references to earlier examples, so you'll at least know when you've encountered something that depends on earlier material. If you do jump from one place to another, these will provide a path back to what you've missed.

Each chapter ends with a special "Tips, Traps, and Tracebacks" section. The tips provide guidance for applying the concepts, mechanisms, and techniques discussed in the chapter. In earlier chapters, many of the tips also provide advice and recommendations for learning Python, using development tools, and organizing programs. The traps are details, warnings, and clarifications regarding common sources of confusion or error for Python programmers (especially new ones). You'll soon learn what a traceback is; for now it is enough to say that they are error messages likely to be encountered when writing code based on the chapter's material.

About Bioinformatics

Any title with the word "bioinformatics" in it is intrinsically ambiguous. There are (at least) three quite different kinds of activities that fall within this term's wide scope. Both the nature of the work performed and the educational backgrounds and technical talents of the people who perform these various activities differ significantly. The three main areas of bioinformatics are:

Computational biology
: Concerned with the development of algorithms for mining biological data and modeling biological phenomena

Software development
: Focused on writing software to implement computational biology algorithms, visualize complex data, and support research and development activity, with particular attention to the challenges of organizing, searching, and manipulating enormous quantities of biological data

Life science research and development
: Focused on the application of the tools and results provided by the other two areas to probe the processes of life

This book is designed to teach you bioinformatics software development. There is no computational biology here: no statistics, formulas, equations—not even explanations of the algorithms that underlie commonly used informatics software. The book's examples are all based on the kind of data life science researchers work with and what they do with it.

The book focuses on practical data management and manipulation tasks. The term "data" has a wide scope here, including not only the contents of databases but also the contents of text files, web pages, and other information sources. Examples focus on genomics, an area that, relative to others, is more mature and easier to introduce to people new to the scientific content of bioinformatics, as well as dealing with data that is more amenable to representation and manipulation in software. Also, and not incidentally, it is the part of bioinformatics with which the author is most familiar.

About the Reader

This book assumes no prior programming experience. Its introduction to and use of Python are completely self-contained. Even if you do have some programming experience, the nature of Python and the book's presentation of technical matter won't necessarily relate directly to anything you've learned before: you too might find much to explore here.

The book also assumes no particular knowledge of or experience in bioinformatics or any of the scientific fields to which it relates. It uses real examples from real biological data, and while nearly all of the topics should be familiar to anyone working in the field, there's nothing conceptually daunting about them. Fundamentally, the goal here is to teach you how to write programs that manipulate data.

This book was written with several audiences in mind:

- Life scientists
- Life sciences students, both undergraduate and graduate
- Technical staff supporting life science research
- Software developers interested in the use of Python in the life sciences

To each of these groups, I offer an introductory message:

Scientists
: Presumably you are reading this book because you've found yourself doing, or wanting to do, some programming to support your work, but you lack the computer science or software engineering background to do it as well as you'd like. The book's introduction to Python programming is straightforward, and its

examples are drawn from bioinformatics. You should find the book readable even if you are just curious about programming and don't plan to do any yourself.

Students
: This book could serve as a textbook for a one-semester course in bioinformatics programming or an equivalent independent study effort. If you are majoring in a life science, the technical competence you can gain from this book will enable you to make significant contributions to the projects in which you participate. If you are majoring in computer science or software engineering but are intrigued by bioinformatics, this book will give you an opportunity to apply your technical education in that field. In any case, nothing in the book should be intimidating to any student with a basic background either in one of the life sciences or in computing.

Technical staff
: You're probably already doing some work managing and manipulating data in support of life science research and development, and you may be accustomed to writing small scripts and performing system maintenance tasks. Perhaps you're frustrated by the limits of your knowledge of computing techniques. Regardless, you have developed an interest in the science and technology of bioinformatics. You want to learn more about those fields and develop your skills in working with biological data. Whatever your training and responsibilities, you should find this book both approachable and helpful.

Programmers
: Bioinformatics software differs from most other software in important, though hard to pin down, ways. Python also differs from other programming languages in ways that you will probably find intriguing. This book moves quickly into significant technical material—it does not follow the pattern of a traditional kind of "Programming in..." or "Learning..." or "Introduction to..." book. Though it makes no attempt to provide a bioinformatics primer, the book includes sufficient examples and explanations to intrigue programmers curious about the field and its unusual software needs.

> I would like to point out to computer scientists and experienced software developers who may read this book that some very particular choices were made for the purposes of presentation to its intended audience. At the risk of sounding arrogant, I assure you that these are backed by deep theoretical knowledge, extensive experience, and a full awareness of alternatives. These choices were made with the intention of simplifying technical vocabulary and presenting as clear and uniform a view of Python programming as possible. They also were based on the assumption that most people making use of what they learn in this book will not move on to more advanced programming or large-scale software development.

Some things that will appear strange to anyone with significant programming experience are in reality true to a pure "Pythonic" approach. It is delightful to have the opportunity to write in this vocabulary without the need to accommodate more traditional terminology.

The most significant example of this is that the word "variable" is never used in the context of assignment statements or function calls. Python does not assign values to variables in the way that traditional "values in a box" languages do. Instead, like some of the languages that influenced its design, what Python does is assign names to values. The assignment statement should be read from left to right as assigning a name to an existing value. This is a very real distinction that goes beyond the ways languages such as Java and C++ refer to objects through pointer-valued variables.

Another aspect of the book's heavily Pythonic approach is its routine use of comprehensions. Approached by someone familiar with other languages, these can appear quite mysterious. For someone learning Python as a first language, though, they can be more natural and easier to use than the corresponding combinations of assignments, tests, and loops or iterations.

Python

This section introduces the Python language and gives instructions for installing and running Python on your machine.

Some Context

There are many kinds of programming languages, with different purposes, styles, intended uses, etc. Professional programmers often spend large portions of their careers working with a single language, or perhaps a few similar ones. As a result, they are often unaware of the many ways and levels at which programming languages can differ. For educational and professional development purposes, it can be extremely valuable for programmers to encounter languages that are fundamentally different from the ones with which they are familiar.

The effects of such an encounter are similar to learning a foreign human language from a different culture or language family. Learning Portuguese when you know Spanish is not much of a mental stretch. Learning Russian when you are a native English speaker is. Similarly, learning Java is quite easy for experienced C++ programmers, but learning Lisp, Smalltalk, ML, or Perl would be a completely different experience.

Broadly speaking, programming languages embody combinations of four paradigms. Some were designed with the intention of staying within the bounds of just one, or perhaps two. Others mix multiple paradigms, although in these cases one is usually dominant. The paradigms are:

Procedural

This is the traditional kind of programming language in which computation is described as a series of steps to be executed by the computer, along with a few mechanisms for branching, repetition, and subroutine calling. It dates back to the earliest days of computing and is still a core aspect of most modern languages, including those designed for other paradigms.

Declarative

Declarative programming is based on statements of facts and logical deduction systems that derive further facts from those. The primary embodiment of the logic programming paradigm is Prolog, a language used fairly widely in Artificial Intelligence (AI) research and applications starting in the 1980s. As a purely logic-based language, Prolog expresses computation as a series of predicate calculus assertions, in effect creating a puzzle for the system to solve.

Functional

In a purely functional language, all computation is expressed as function calls. In a truly pure language there aren't even any variable assignments, just function parameters. Lisp was the earliest functional programming language, dating back to 1958. Its name is an acronym for "LISt Processing language," a reference to the kind of data structure on which it is based.

Lisp became the dominant language of AI in the 1960s and still plays a major role in AI research and applications. The language has evolved substantially from its early beginnings and spawned many implementations and dialects, although most of these disappeared as hardware platforms and operating systems became more standardized in the 1980s.

A huge standardization effort combining ideas from several major dialects and a great many extensions, including a complete object-oriented (see below) component, was undertaken in the late 1980s. This effort resulted in the now-dominant CommonLisp.* Two important dialects with long histories and extensive current use are Scheme and Emacs Lisp, the scripting language for the Emacs editor. Other functional programming languages in current use are ML and Haskell.

Object-oriented

Object-oriented programming was invented in the late 1960s, developed in the research community in the 1970s, and incorporated into languages that spread widely into both academic and commercial environments in the 1980s (primarily Smalltalk, Objective-C, and C++). In the 1990s this paradigm became a key part of modern software development approaches. Smalltalk and Lisp continued to be used, C++ became dominant, and Java was introduced. Mac OS X, though built on a Unix-like kernel, uses Objective-C for upper layers of the system, especially the user interface, as do applications built for Mac OS X. JavaScript, used primarily to program web browser actions, is another object-oriented language. Once a

* See *http://www.lispworks.com/documentation/HyperSpec/Body/01_ab.htm*.

radical innovation, object-oriented programming is today very much a mainstream paradigm.

Another dimension that distinguishes programming languages is their primary intended use. There have been languages focused on string matching, languages designed for embedded devices, languages meant to be easy to learn, languages built for efficient execution, languages designed for portability, languages that could be used interactively, languages based largely on list data structures, and many other kinds.

Language designers, whether consciously or not, make choices in these and other dimensions. Subsequent evolutions of their languages are subject to market forces, intellectual trends, hardware developments, and so on. These influences may help a language mature and reach a wider audience. They may also steer the language in directions somewhat different from those originally intended.

The Python Language

Simply put, Python is a beautiful language. It is effective for everything from teaching new programmers to advanced computer science study, from simple scripts to sophisticated advanced applications. It has always had some purchase in bioinformatics, and in recent years its popularity has been increasing rapidly. One goal of this book is to help significantly expand Python's use for bioinformatics programming.

Python features a syntax in which the ends of statements are marked only by the end of a line, and statements that form part of a compound statement are indented relative to the lines of code that introduce them. The semicolons or keywords that end statements and the braces that group statements in other languages are entirely absent.

Programmers familiar with "standard syntax" languages often find Python's uncluttered syntax deeply disconcerting. New programmers have no such problem, and for them, this simple and readable syntax is far easier to deal with than the visually arcane constructions using punctuation (with the attendant compilation errors that must be confronted). Traditional programmers should reconsider Python's syntax after performing this experiment:

1. Open a file containing some well-formatted code.
2. Delete all semicolons, braces, and terminal keywords such as `end`, `endif`, etc.
3. Look at the result.

To the human eye, the simplified code is easier to read—and it looks an awful lot like Python. It turns out that the semicolons, terminal keywords, and braces are primarily for the benefit of the compiler. They are not really necessary for human writers and readers of program code. Python frees the programmer from the drudgery of serving as a compiler assistant.

Python is an interesting and powerful language with respect to computing paradigms. Its skeleton is procedural, and it has been significantly influenced by functional

programming, but it has evolved into a fundamentally object-oriented language. (There is no declarative programming component—of the four paradigms, declarative programming is the one least amenable to fitting together with another.) Few, if any, other languages provide a blend like this as seamlessly and elegantly as does Python.

Installing Python

This book uses Python 3, the language's first non-backward-compatible release. With a few minor changes, noted where applicable, Python 2.x will work for most of the book's examples. There are a few notes about Python 2 in Chapters 1, 3, and 5; they are there not just to help you if you find yourself using Python 2 for some work, but also for when you read Python 2 code. The major exception is that `print` was a statement in Python 2 but is now a function, allowing for more flexibility. Also, Python 3 reorganized and renamed some of its library modules and their contents, so using Python 2.x with examples that demonstrate the use of certain modules would involve more than a few minor changes.

Determing Which Version of Python Is Installed

Some version of Python 2 is probably installed on your computer, unless you are using Windows. Typing the following into a command-line window (using `%` as an example of a command-line prompt) will tell you which version of Python is installed as the program called *python*:

```
% python -V
```

The name of the executable for Python 3 may be *python3* instead of just *python*. You can type this:

```
% python3 -V
```

to see if that is the case.

If you are running Python in an integrated development environment—in particular IDLE, which is part of the Python installation—type the following at the prompt (>>>) of its interactive shell window to get information about its version:

```
>>> from sys import version
>>> version
```

If this shows a version earlier than 3, look for another version of the IDE on your computer, or install one that uses Python 3. (The Python installation process installs the GUI-based IDLE for whatever version of Python is being installed.)

The current release of Python can be downloaded from *http://python.org/download/*. Installers are available for OS X and Windows. With most distributions of Linux, you should be able to install Python through the usual package mechanisms. (Get help from someone who knows how to do that if you don't.) You can also download the source,

unpack the archive, and, following the steps in the "Build Instructions" section of the *README* file it contains, configure, "make," and install the software.

> If you are installing Python from its source code, you may need to download, configure, make, and install several libraries that Python uses if available. At the end of the "make" process, a list of missing optional libraries is printed. It is not necessary to obtain all the libraries. The ones you'll want to have are:
>
> - `curses`
> - `gdbm`
> - `sqlite3`[†]
> - Tcl/Tk[‡]
> - `readline`
>
> All of these should be available through standard package installers.

Running Python

You can start Python in one of two ways:

1. Type `python3` on the command line.[§]
2. Run an IDE. Python comes with one called IDLE, which is sufficient for the work you'll do in this book and is a good place to start even if you eventually decide to move on to a more sophisticated IDE.

The term *Unix* in this book refers to all flavors thereof, including Linux and Mac OS X. The term *command line* refers to where you type commands to a "shell"—in particular, a Unix shell such as `tcsh` or `bash` or a Windows command window—as opposed to typing to the Python interpreter. The term *interpreter* may refer to either the interpreter running in a shell, the "Python Shell" window in IDLE, or the corresponding window in whatever other development environment you might be using.

† You can find precompiled binaries for most platforms at *http://sqlite.org/download.html*.

‡ See *http://www.activestate.com/activetcl*.

§ On OS X, a command-line shell is obtained by running the Terminal application, found in the *Utilities* folder in the *Applications* folder. On most versions of Windows, a "Command Prompt" window can be opened either by selecting Run from the Start menu and typing `cmd` or by selecting Accessories from the Start menu, then the Command Prompt entry of that menu. You may also find an Open Command Line Here entry when you right-click on a folder in a Windows Explorer window; this is perhaps the best way to start a command-line Python interpreter in Windows because it starts Python with the selected folder as the current directory. You may have to change your path settings to include the directory that contains the Python executable file. On a Unix-based system, you do that in the "rc" file of the shell you are using (e.g., *~/.bashrc*). On Windows, you need to set the path as an environment variable, a rather arcane procedure that differs among different versions of Windows. You can also type the full path to the Python executable; on Windows, for example, that would probably be *C:\\python3.1\\python.exe*.

When Python starts interactively, it prints some information about its version. Then it repeats a cycle in which it:

1. Prints the *prompt* `>>>` to indicate that it is waiting for you to type something
2. Reads what you type
3. Interprets its meaning to obtain a value
4. Prints that value

Throughout the book, the appearance of the `>>>` prompt in examples indicates the use of the interpreter. Like nearly all command-line interactive applications, the Python interpreter won't pay any attention to what you've typed until you press the Return (Enter) key. Pressing the Return key does not always provide complete input for Python to process, though; as you'll see, it is not unusual to enter multiline inputs. In the command-line interpreter, Python will indicate that it is still waiting for you to complete your input by starting lines following the `>>>` prompt with `...`. IDLE, unfortunately, gives no such indication.

Both IDLE and the command-line interpreter provide simple keyboard shortcuts for editing the current line before pressing Return. There are also keys to recall previous inputs. IDLE provides quite a few additional keyboard shortcuts that are worth learning early on. In addition, if you are using an IDE—IDLE, in particular—you'll be able to use the mouse to click to move the cursor on the input line.

To get more information about using IDLE, select "IDLE Help" from its Help menu. That won't show you the keyboard shortcuts, though; they are listed in the "Keys" tab of IDLE's preferences dialog. Note that you can use that dialog to change the keystroke assignments, as well as the fonts, colors, window size, and so on.

When you want to quit a command-line Python interpreter, simply type Ctrl-D in Unix (including Linux and the Mac OS X Terminal application). In Windows, type Ctrl-Z. You exit an IDE with the usual Quit menu command.

Notes

I end this preface with some notes about things I think will help you make the most of your experience with this book.

Reading and Reference Recommendations

The documentation that comes with the Python installation is excellent, extensive, and well organized, but it can be overwhelming to newcomers. Moreover, the topics this book presents and the way it presents them are designed specifically with bioinformatics students and professionals in mind (though of course it's hoped that it will be valuable to a much wider audience than that). Unless you find yourself needing more information about the Python language or library than is provided in this book while

you're reading it, it's probably best to wait until you finish it before spending much time with the documentation. The documentation is aimed primarily at Python programmers. You'll be one when you finish this book, at which point you'll use the documentation all the time.

With respect to the bioinformatics side of things, I trust you won't encounter anything unfamiliar here. But if you do, or you want to delve deeper, Wikipedia is a remarkably deep resource for bioinformatics—for programming and computer science too, for that matter. There are two astoundingly extensive bioinformatics references you should at least have access to, if not actually own:

- *Bioinformatics and Functional Genomics*, Second Edition, by Jonathan Pevsner (Wiley-Blackwell)
- *Bioinformatics: Sequence and Genome Analysis*, Second Edition, by David W. Mount (Cold Spring Harbor Laboratory Press)

An unusual collection of essays containing detailed information about approaches to analyzing bioinformatics data and the use of a vast array of online resources and tools is:

- *Bioinformatics for Geneticists: A Bioinformatics Primer for the Analysis of Genetic Data*, Second Edition, by Michael R. Barnes (Ed.) (Wiley)

Example Code

All the code for this book's examples, additional code, some lists of URLs, data for the examples, and so forth are found on the book's website. In many cases, there is a sequence of files for the same example or set of examples that shows the evolution of the example from its simplest start to where it ends up. In a few cases, there are versions of the code that go beyond what is shown in the book. There are also some examples that are not in the book at all, as well as exercises for each chapter.

Within the book's code examples, statement keywords are in boldface. Comments and documentation are in serif typeface. Some examples use oblique monospace to indicate descriptive "pseudocode" meant to be replaced with actual Python code. A shaded background indicates either code that has changed from the previous version of an example or code that implements a point made in the preceding paragraph(s).

Unfortunate and Unavoidable Vocabulary Overlap

This book's vocabulary is drawn from three domains that, for the most part, are independent of each other: computer science (including data structures and programming language concepts), Python (which has its own names for the program and data structures it offers), and biology. Certain words appear in two or even all three of these domains, often playing a major role and having a different meaning in each. The result is unfortunate and unavoidable collisions of vocabulary. Care was taken to establish

sufficient context for the use of these words to make their meanings clear, but the reader should be prepared for the occasional mental backtrack to try another meaning for a term while attempting to make sense of what is being said.

Even within a single domain, a term's importance does not necessarily rescue it from ambiguity. Consider the almost unrelated meanings of the term "frame" as the offset from the start of a DNA sequence and in the phrase "open reading frame." There can be many open reading frames in a frame and many frames with open reading frames. And sometimes there are three frames to consider, and sometimes also the reverse complement frames, making six. Open reading frames can appear in any of the six reading frames.

The vocabulary overlap is so omnipresent that it is almost humorous. Then again, the words involved are fine words that have meanings in a great many other domains too, so we should not be surprised to encounter them in our three. Even though you have not yet been properly introduced to them, Table P-1 lists some of the most vexing examples. Stay particularly alert when you encounter these, especially when you see the words in code examples.

Table P-1. Domain-ambiguous terms

Term	Biology	Programming	Python
Sequence	Part of a DNA or RNA molecule; more often refers to the abstraction thereof, as represented with letters	(Usually) one of a number of data structures that arrange their elements linearly	A linear, and therefore numerically indexable, collection of values
Base	A single nucleotide in a DNA or RNA molecule	Base 10, 16, 2, etc.	Base 10, 16, 2, etc., as used in input and output operations
String	A series of letters representing a DNA, RNA, or amino acid sequence	A sequence of characters, often a "primitive type" of a language	An immutable sequence type named `str`
Expression	The production of proteins under the control of cellular machinery influenced by life stage, the organ containing the cell, internal states (disease, hunger), and external conditions (dryness, heat)	(1) (Generally) a combination of primitive values, operators, and function calls, with specifics differing significantly among languages	(1) A combination of primitive values, operators, and function calls
		(2) Regular expression: a pattern describing a set of strings with notations for types of characters, grouping, repetition, and so on, the details of which differ among languages and editors	(2a) A regular expression string (2b) A regular expression string compiled into a regular expression object
Type	The specimen of an organism first used to describe and name it	A theoretical construct defined differently in different contexts and implemented differently by different programming lan-	Synonymous with "class," but often used in the context of Python's built-in types, as op-

Term	Biology	Programming	Python
		guages; corresponds roughly to "the kind of thing" something is and "the kind of operations" it supports	posed to classes defined in a Python or externally obtained library or in user code
Translate, `translate`	Convert DNA codons (base triples) to amino acids according to the genetic code of the organism	Convert computer code in one language into computer code in another, typically lower-level, language	A method of `str` that uses a table to produce a new `str` with all the characters of the original replaced by the corresponding entries in the table
Class	One of the levels in the standard taxonomic classification of organisms	In languages that support object-oriented programming, the encapsulated definition of data and related code	As in programming; more specifically, the type of an object, which itself is an object that defines the methods for its instances
Loop	A property of RNA secondary structures (among other meanings)	An action performed repeatedly until some condition is no longer true	An action performed repeatedly until some condition is no longer true
Library	A collection of related sequences, most commonly used in the context of a library of expressed RNA in cDNA form	Like a program, but meant to be used by other programs rather than as a free-standing application; most languages use a core set of libraries and provide a large selection of optional ones	A collection of modules, each containing a collection of related definitions, as in "Python comes with an extensive library of optional tools and facilities"
Complement	The nucleotide with which another always pairs	"Two's complement" is the standard representation of negative integers	
R.E.	Restriction enzyme		Regular expression

Fortunately, while the term "sequence" has a conceptual meaning in Python, there is nothing defined in the language by that name, so we can use it in our descriptions and code examples. Likewise, the name of the string type is `str`, so we can use the term "string" in descriptions and examples. The lack of overlap in these instances saves a fair amount of awkward clarification that would otherwise be required.

Comments

Write code as you read: it will really help you understand and benefit from what you are reading. Read through the code examples. Look for more detailed code, additional examples, and exercises on the book's website.

Bioinformatics is a fascinating field. Python is a wonderful language. Programming is an exciting challenge. Technologies covered here are important. This book is an invitation to investigate, experience, and learn (more) about all of these topics. Enjoy!

Conventions Used in This Book

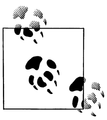
The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, filenames, and file extensions, and is used for documentation and comments in code examples

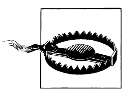
`Constant width`
: Used for names of values, functions, methods, classes, and other code elements, as well as the code in examples, the contents of files, and printed output

`Constant width bold`
: Indicates user input to be typed at an interactive prompt, and highlights Python statement keywords in examples

`Constant width italic`
: Shows text that should be replaced with user-supplied values; also used in examples for "pseudocode" meant to be replaced with actual Python code



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Some chapters include text and code in specially labeled sidebars. These capture explanations and examples in an abstract form. They are meant to serve as aids to learning the first time you go through the material, and useful references later. There are three kinds of these special sidebars:

> **STATEMENT**
>
> Every kind of Python statement is explained in text and examples and summarized in a statement box.

> **SQL**
>
> SQL boxes are used in Chapter 10 to summarize statements from the database language SQL.

TEMPLATE

In place of theoretical discussions of the many ways a certain kind of code can be written, this book presents generalized abstract patterns that show how such code would typically appear in bioinformatics programs. The idea is that, at least in the early stages of using what you learn in this book, you can use the templates in your programs, replacing the abstract parts with the concrete details of your code.

We'd Like to Hear from You

Every example in this book has been tested, but occasionally you may encounter problems. Mistakes and oversights can occur, and we will gratefully receive details of any that you find, as well as any suggestions you would like to make for future editions. You can contact the author and editor at:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596154509/

To comment or ask technical questions about this book, send email to the following, quoting the book's ISBN number (9780596154509):

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example

code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Bioinformatics Programming using Python* by Mitchell L Model. Copyright 2010 Mitchell L Model, 978-0-596-15450-9."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

Acknowledgments

The many O'Reilly people who worked with me to turn my draft into a book were an impressive lot. Two of them deserve particular thanks. The term "editor" does not come close to describing the roles Mike Loukides played in this project, among them manager, confidante, and contributor; it was delightful to work with him and to have him as an audience for my technical musings. Rachel Head, copyeditor extraordinaire, contributed extensively to the clarity and accuracy of this book; I enjoyed working with her and was amazed by her ability to detect even tiny technical inconsistencies.

My thanks to James Tisdall, whose O'Reilly books *Beginning Perl for Bioinformatics* and *Mastering Perl for Bioinformatics* were the original impetus—though longer ago than I would like to remember—for my writing a similar book using Python and whose encouragements I much appreciated. A number of reviewers made helpful comments. Foremost among them was my friend and colleague Tom Stambaugh, founder of Zeetix LLC, who gave one draft an extremely close reading that resulted in many changes after hours of discussion. Though I initially resisted much of what reviewers suggested, I eventually acceded to most of it, which considerably improved the book.

I thank my students at Northeastern University's Professional Masters in Bioinformatics program for their patience, suggestions, and error detection while using earlier versions of the book's text and code. Special thanks go to Jyotsna Guleria, a graduate of that program, who wrote test programs for the example code that uncovered significant latent errors. (Extended versions of the test programs can be found on the book's website.) Finally, I hope what I have produced justifies what my friends, family, and colleagues endured during its creation—especially Janet, my wife, whose unwavering support during the book's writing made the project possible.

CHAPTER 1

Primitives

Computer programs manipulate data. This chapter describes the simplest kinds of Python data and the simplest ways of manipulating them. An individual item of data is called a *value*. Every value in Python has a *type* that identifies the *kind* of value it is. For example, the type of `2` is `int`. You'll get more comfortable with the concepts of types and values as you see more examples.

The Preface pointed out that Python is a multiparadigm programming language. The terms "type" and "value" come from traditional procedural programming. The equivalent object-oriented terms are *class* and *object*. We'll mostly use the terms "type" and "value" early on, then gradually shift to using "class" and "object" more frequently. Although Python's history is tied more to object-oriented programming than to traditional programming, we'll use the term *instance* with both terminologies: each value is an instance of a particular type, and each object is an instance of a particular class.

Simple Values

Types for some simple kinds of values are an integral part of Python's implementation. Four of these are used far more frequently than others: *logical* (Boolean), *integer*, *float*, and *string*. There is also a special no-value value called `None`.

When you enter a value in the Python interpreter, it prints it on the following line:

```
>>> 90
90
>>>
```

When the value is `None`, nothing is printed, since `None` means "nothing":

```
>>> None
>>>
```

If you type something Python finds unacceptable in some way, you will see a multiline message describing the problem. Most of what this message says won't make sense

until we've covered some other topics, but the last line should be easy to understand and you should learn to pay attention to it. For example:

```
>>> Non
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    Non
NameError: name 'Non' is not defined
>>>
```

When a `#` symbol appears on a line of code, Python ignores it and the rest of the line. Text following the `#` is called a *comment*. Typically comments offer information about the code to aid the reader, but they can include many other kinds of text: a programmer's notes to fix or investigate something, a reference (documentation entry, book title, URL, etc.), and so on. They can even be used to "comment out" code lines that are not working or are obsolete but still of interest. The code examples that follow include occasional comments that point out important details.

Booleans

There are only two Boolean values: `True` and `False`. Their type is `bool`. Python names are "case-sensitive," so `true` is not the same as `True`:

```
>>> True
True
>>> False
False
```

Integers

There's not much to say about Python integers. Their type is `int`, and they can have as many digits as you want. They may be preceded by a plus or minus sign. Separators such as commas or periods are not used:

```
>>> 14
14
>>> -1
-1
>>> 1112223334445556667778889990000000000000          # a very large integer!
1112223334445556667778889990000000000000
```

Python 2: A distinction is made between integers that fit within a certain (large) range and those that are larger; the latter are a separate type called `long`.

Integers can also be entered in *hexadecimal notation*, which uses base 16 instead of base 10. The letters A through F represent the hexadecimal digits 10 through 15. Hexadecimal notation begins with `0x`. For example:

```
>>> 0x12                    # (1 x 16 )+ 2
18
>>> 0xA40                   # (10 x 16 x 16) + (4 x 16) + 0
2624
>>> 0xFF                    # (15 x 16) + 15
255
```

The result of entering a hexadecimal number is still an integer—the only difference is in how you write it. Hexadecimal notation is used in a lot of computer-related contexts because each hexadecimal digit occupies one half-byte. For instance, colors on a web page can be specified as a set of three one-byte values indicating the red, green, and blue levels, such as `FFA040`.

Floats

"Float" is an abbreviated version of the term "floating point," which refers to a number that is represented in computer hardware in the equivalent of scientific notation. Such numbers consist of two parts: digits and an exponent. The exponent is adjusted so the decimal point "floats" to just after the first digit (or just before, depending on the implementation), as in scientific notation.

The written form of a `float` always contains a decimal point and at least one digit after it:

```
>>> 2.5
2.5
```

You might occasionally see floats represented in a form of scientific notation, with the letter "e" separating the base from the exponent. When Python prints a number in scientific notation it will always have a single digit before the decimal point, some number of digits following the decimal point, a `+` or `-` following the `e`, and finally an integer. On input, there can be more than one digit before the decimal point. Regardless of the form used when entering a float, Python will output very small and very large numbers using scientific notation. (The exact cutoffs are dependent on the Python implementation.) Here are some examples:

```
>>> 2e4                     # Scientific notation, but...
20000.0                     # within the range of ordinary floats.
>>> 2e-2
0.02
>>>.0001                    # Within the range of ordinary floats
0.0001                      # so printed as an ordinary float.
>>>.00001                   # An innocent-looking float that is
1e-05                       # smaller than the lower limit, so e.
>>> 1002003004005000.       # A float with many digits that is
1002003004005000.0          # smaller than the upper limit, so no e.
>>> 100200300400500060.     # Finally, a float that is larger than the
1.0020030040050006e+17      # upper limit, so printed with an e.
```

Strings

Strings are series of Unicode[*] characters. Their type is `str`. Many languages have a separate "character" type, but Python does not: a lone character is simply a string of length one. A string is enclosed in a pair of single or double quotes. Other than style preference, the main reason to choose one or the other kind of quote is to make it convenient to include the other kind inside a string.

If you want a string to span multiple lines, you must enclose it in a matched pair of three single or double quotes. Adding a backslash in front of certain characters causes those characters to be treated specially; in particular, `'\n'` represents a line break and `'\t'` represents a tab.

> **Python 2:** Strings are composed of one-byte characters, not Unicode characters; there is a separate string type for Unicode, designated by preceding the string's opening quote with the character `u`.

We will be working with strings a lot throughout this book, especially in representing DNA/RNA base and amino acid sequences. Here are the amino acid sequences for some unusually small bacterial restriction enzymes:[†]

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'
MNKMDLVADVAEKTDLSKAKATEVIDAVFA
>>> "AARHQGRGAPCGESFWHWALGADGGHGHAQPPFRSSRLIGAERQPTSDCRQSLQQSPPC"
AARHQGRGAPCGESFWHWALGADGGHGHAQPPFRSSRLIGAERQPTSDCRQSLQQSPPC
>>> """MKQLNFYKKN SLNNVQEVFS YFMETMISTN RTWEYFINWD KVFNGADKYR NELMKLNSLC GS
LFPGEELK SLLKKTPDVV KAFPLLLAVR DESISLLD"""
'MKQLNFYKKN SLNNVQEVFS YFMETMISTN RTWEYFINWD KVFNGADKYR NELMKLNSLC GS
LFPGEELK\nSLLKKT PDVV KAFPLLLAVR DESISLLD'
>>> '''MWNSNLPKPN AIYVYGVANA NITFFKGSDI LSYETREVLL KYFDILDKDE RSLKNALKD LEN PFGFAPYI
RKAYEHKRNF LTTTRLKASF RPTTF'''
'MWNSNLPKPN AIYVYGVANA NITFFKGSDI LSYETREVLL KYFDILDKDE RSLKNALKDL EN\nPFGF
APYI RKAYEHKRNF LTTTRLKASF RPTTF'
```

There are three situations that cause input or output to begin on a new line:

- You hit Return as you are typing inside a triple-quoted string.
- You keep typing characters until they "wrap around" to the next line before you press Return.
- The interpreter responds with a string that is too long to fit on one line.

[*] Unicode characters occupy between one and four bytes each in memory, depending on several factors. See *http://docs.python.org/3.1/howto/unicode.html* for details (in particular, *http://docs.python.org/3.1/howto/unicode.html#encodings*). For general information about Unicode outside of Python, consult *http://www.unicode.org/standard/WhatIsUnicode.html*, *http://www.unicode.org/standard/principles.html*, and *http://www.unicode.org/resources*.

[†] Data for these examples was obtained from the "Official REBASE Homepage" site. Files in formats used by various applications can be downloaded from *http://rebase.neb.com/rebase/rebase.files.html*.

Only the first one is "real." The other two are simply the effect of output "line wrapping" like what you would see in text editors or email programs. In the second and third situations, if you change the width of the window the input and output strings will be "rewrapped" to fit the new width. The first case does *not* cause a corresponding line break when the interpreter prints the string—the Return you typed becomes a `'\n'` in the string.

Normally, Python uses a pair of single quotes to enclose strings it prints. However, if the string contains single quotes (and no double quotes), it will use double quotes. It never prints strings using triple quotes; instead, the line breaks typed inside the string become `'\n'`s.

Expressions

An *operator* is a symbol that indicates a calculation using one or more *operands*. The combination of the operator and its operand(s) is an *expression*.

Numeric Operators

A *unary operator* is one that is followed by a single operand. A *binary operator* is one that appears between two operands. It isn't necessary to surround operators with spaces, but it is good style to do so. Incidentally, when used in a numeric expression, `False` is treated as `0` and `True` as `1`.

Plus and minus can be used as either unary or binary operators:

```
>>> -1                  # unary minus
-1
 >>> 4 + 2
6
>>> 4 - 1
3
>>> 4 * 3
12
```

The power operator is ** (i.e., n^k is written `n ** k`):

```
>>> 2 ** 10
1024
```

There are three operators for the division of one integer by another: `/` produces a float, `//` (*floor division*) an integer with the remainder ignored, and `%` (*modulo*) the remainder of the floor division. The formal definition of floor division is "the largest integer not greater than the result of the division":

```
>>> 11 / 4
2.75
>>> 11 // 4             # "floor" division
2
```

```
>>> 11 % 4              # remainder of 11 // 3
3
```

Python 2: The `/` operator performs floor division when both operands are `int`s, but ordinary division if one or both operands are `float`s.

Whenever one or both of the operators in an arithmetic expression is a float, the result will be a float:

```
>>> 2.0 + 1
3.0
>>> 12 * 2.5
30.0
>>> 7.5 // 2
3.0
```

While the value of floor division is *equal* to an integer value, its *type* may *not* be integer! If both operands are `int`s, the result will be an `int`, but if either or both are `float`s, the result will be a `float` that represents an integer.

The result of an operation does not always print the way you might expect. Consider the following numbers:

```
>>> .009
.009
>>> .01
.01
>>> .029
.029
>>> .03
.03
>>> .001
.001
```

So far, everything is as expected. If we subtract the first from the second and the third from the fourth, we should in both cases get the result `.001`. Typing in `.001` also gives the expected result. However, typing in the subtraction operations does not:

```
>>> .03 - .029
0.0009999999999999974
>>> .01 - .009
0.0010000000000000009
```

Strange results like this arise from two sources:

- For a given base, only some rational numbers have "exact" representations—that is, their decimal-point representations terminate after a finite number of digits. The rest end in an infinitely repeating sequence of digits (e.g., `1/3 = 0.3333333...`).

- A computer stores rational numbers in a finite number of binary digits; the binary representation of a rational number may in fact have an exact binary representation, but one that would require more digits than are used.

> A *rational number* is one that can be expressed as `a/b`, where `b` is not zero; the decimal-point expression of a rational number in a given number system either has a finite number of digits or ends with an infinitely repeating sequence of digits. There's nothing wrong with the binary system: whatever base is used, some real numbers have exact representations and others don't. Just as only some rational numbers have exact decimal representations, only some rational numbers have exact binary representations.

As you can see from the results of the two division operations, the difference between the ideal rational number and its actual representation is quite small, but in certain kinds of computations the differences do accumulate.‡

Here's an early lesson in an extremely important programming principle: *don't trust what you see!* Everything printed in a computing environment or by a programming language is an interpretation of an internal representation. That internal representation may be manipulated in ways that are intended to be helpful but can be misleading. In the preceding example, `0.009` in fact does *not* have an exact binary representation. In Python 2, it would have printed as `0.0089999999999999993`, and `0.003` would have printed as `0.0089999999999999993`. The difference is that Python 3 implements a more sophisticated printing mechanism for rational numbers that makes some of them *look* as they would have had you typed them.

Logical Operations

Python, like other programming languages, provides operations on "truth values." These follow the mathematical laws of *Boolean logic*. The classic Boolean operators are `not`, `and`, and `or`. In Python, those are written just that way rather than using special symbols:

```
>>> not True
False
>>> not False
True
>>> True and True
True
>>> True and False
False
>>> True or True
True
```

‡ A computer science field called "numerical analysis" provides techniques for managing the accumulation of such errors in complex or repetitive computations.

```
>>> True or False
True
>>> False and False
False
>>> False or True
```

The results of `and` and `or` operations are not converted to Booleans. For `and` expressions, the first operand is returned if it is false; otherwise, the second operand is returned. For `or` expressions, the first operand is returned if it is true; otherwise, the second operand is returned. For example:

```
>>> '' and 'A'
''                     # Not False: '' is a false value
>>> 0 and 1 or 2       # Read as (0 and 1) or 2
2                      # Not True: 2 is a false value
```

While confusing, this can be useful; we'll see some examples later.

The operands of `and` and `or` can actually be anything. `None`, `0`, `0.0`, and the empty string, as well as the other kinds of "empty" values explained in Chapter 3, are considered `False`. Everything else is treated as `True`.

To avoid repetition and awkward phrases, this book will use "true" and "false" in regular typeface to indicate values considered to be `True` and `False`, respectively. It will only use the capitalized words `True` and `False` in the code typeface when referring to those specific Boolean values.

There is one more logical operation in Python that forms a *conditional expression*. Written using the keywords `if` and `else`, it returns the value following the `if` when the condition is true and the value following the `else` when it is false. We'll look at some more meaningful examples a bit later, but here are a few trivial examples that show what conditional expressions look like:

```
>>> 'yes' if 2 - 1 else 'no'
'yes'
>>> 'no' if 1 % 2  else 'no'
'no'
```

In addition to the Boolean operators, there are six *comparison operators* that return Boolean values: `==`, `!=`, `<`, `<=`, `>`, and `>=`. These work with many different kinds of operands:

```
>>> 2 == 5 // 2
True
>>> 3 > 13 % 5
False
>>> 'one' < 'two'
True
```

```
>>> 'one' != 'one'
False
```

You may already be familiar with logical and comparison operations from other computer work you've done, if only entering spreadsheet formulas. If these are new to you, spend some time experimenting with them in the Python interpreter until you become comfortable with them. You will use them frequently in code you write.

String Operations

There are four binary operators that act on strings: `in`, `not in`, `+`, and `*`. The first three expect both operands to be strings. The last requires the other operator to be an integer. A one-character substring can be extracted with *subscription* and a longer substring by *slicing*. Both use square brackets, as we'll see shortly.

String operators

The `in` and `not in` operators test whether the first string is a substring of the second one (starting at any position). The result is `True` or `False`:

```
>>> 'TATA' in 'TATATATATATATATATATATATA'
True
>>> 'AA' in 'TATATATATATATATATATATATA'
False
>>> 'AA' not in 'TATATATATATATATATATATATA'
True
```

A new string can be produced by *concatenating* two existing strings. The result is a string consisting of all the characters of the first operand followed by all the characters of the second. Concatenation is expressed with the plus operator:

```
>>> 'AC' + 'TG'
'ACTG'
>>> 'aaa' + 'ccc' + 'ttt' + 'ggg'
'aaaccctttggg'
```

A string can be repeated a certain number of times by multiplying it by an integer:

```
>>> 'TA' * 12
'TATATATATATATATATATATATA'
>>> 6 * 'TA'
'TATATATATATA'
```

Subscription

Subscription extracts a one-character substring of a string. Subscription is expressed with a pair of square brackets enclosing an integer-valued expression called an *index*. The first character is at position 0, not 1:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[0]
'M'
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[1]
'N'
```

The index can also be negative, in which case the index is counted from the end of the string. The last character is at index `-1`:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[-1]
'A'
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[-5]
'D'
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[7 // 2]
'K'
```

As Figure 1-1 shows, starting at `0` from the beginning or end of a string, an index can be thought of as a label for the character to its right. The end of a string is the position one after the last element. If you are unfamiliar with indexing in programming languages, this is probably an easier way to visualize it than if you picture the indexes as aligned with the characters.

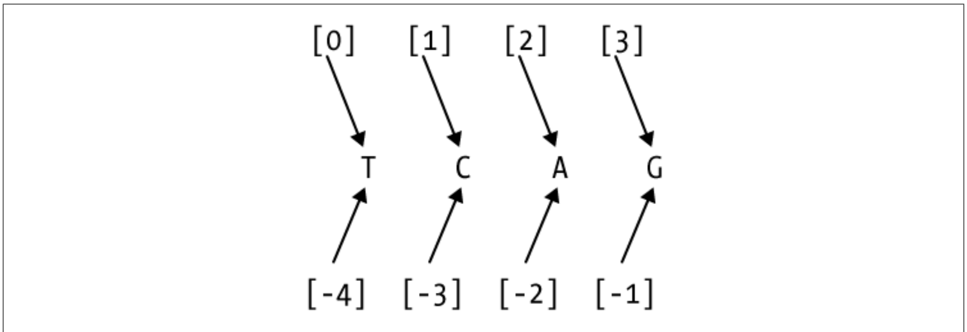


Figure 1-1. Index positions in strings

Attempting to extract a character before the first or after the last causes an error, as shown here:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[50]
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[50]
IndexError: string index out of range
```

The last line reports the nature of the error, while the next-to-last line shows the input that caused the error.

Slicing

Slicing extracts a series of characters from a string. You'll use it often to clearly and concisely designate parts of strings. Figure 1-2 illustrates how it works.

The character positions of a slice are specified by two or three integers inside square brackets, separated by colons. The first index indicates the position of the first character to be extracted. The second index indicates where the slice ends. The character at that position is not included in the slice. A slice `[m:n]` would therefore be read as "from

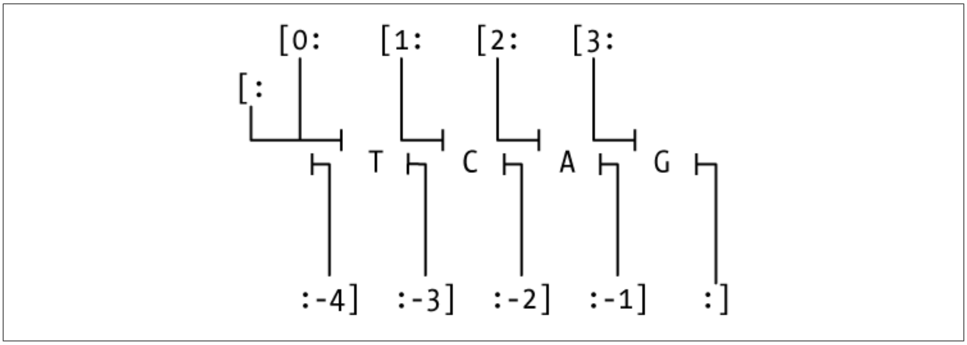


Figure 1-2. String slicing

character `m` up to but not including character `n`." (We'll explore the use of the third index momentarily). Here are a few slicing examples:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[1:4]
'NKM'
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[4:-1]
'DLVADVAEKTDLSKAKATEVIDAVF'
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[-5:-4]
'D'
```

Either of the indexes can be positive, indicating "from the beginning," or negative, indicating "from the end." If neither of the two numbers is negative, the length of the resulting string is the difference between the second and the first. If either (or both) is negative, just add it to the length of the string to convert it to a nonnegative number.

What if the two numbers are the same? For example:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[5:5]
''
```

Since this reads as "from character 5 up to but not including character 5," the result is an empty string. Now, what about character positions that are out of order—i.e., where the first character occurs after the second? This results in an empty string too:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[-4:-6]
''
```

For subscription, the index must designate a character in the string, but the rules for slicing are less constraining.

When the slice includes the beginning or end of the string, that part of the slice notation may be omitted. Note that omitting the second index is not the same as providing `-1` as the second index—omitting the second index says to go up to the end of the string, one past the last character, whereas `-1` means go up to the penultimate character (i.e., up to but not including the last character):

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[:8]
'MNKMDLVADVAEKTDLSKAKAT'
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[9:]
```

```
'VAEKTDLSKAKATEVIDAVFA'
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[9:-1]
'VAEKTDLSKAKATEVIDAVF'
```

In fact, both indexes can be omitted, in which case the entire string is selected:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[:]
'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'
```

Finally, as mentioned earlier, a slice operation can specify a third number, also following a colon. This indicates a number of characters to skip after each one that is included, known as a *step*. When the third number is omitted, as it often is, the default is `1`, meaning don't skip any. Here's a simple example:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[0:9:3]
'MMV'
```

This example's result was obtained by taking the first, fourth, and seventh characters from the string. The step can be also be a negative integer. When the step is negative, the slice takes characters in reverse order. To get anything other than an empty string when you specify a negative step, the start index must be greater than the stop index:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[16:0:-4]
'SKDD'
```

Notice that the first character of the string is not included in this example's results. The character at the stop index is never included. Omitting the second index so that it defaults to the beginning of the string—beginning, not end, because the step is negative—results in a string that does include the first character, assuming the step would select it. Changing the previous example to omit the `0` results in a longer string:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[16::-4]
'SKDDM'
```

Omitting the first index when the step is negative means start from the end of the string:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[:25:-1]
'AFVA'
```

A simple but nonobvious slice expression produces a reversed copy of a string: `s[::-1]`. This reads as "starting at the end of the string, take every character up to and including the first, in reverse order":

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[::-1]
'AFVADIVETAKAKSLDTKEAVDAVLDMKNM'
```

Calls

We'll look briefly at calls here, deferring details until later. A *call* is a kind of expression.

Function calls

The simplest kind of call invokes a *function*. A call to a function consists of a *function name*, a pair of parentheses, and zero or more *argument* expressions separated by

commas. The function is *called*, does something, then *returns* a value. Before the function is called the argument expressions are evaluated, and the resulting values are *passed* to the function to be used as input to the computation it defines. An argument can be any kind of expression whose result has a type acceptable to the function. Those expressions can also include function calls.

Each function specifies the number of arguments it is prepared to receive. Most functions accept a fixed number—possibly zero—of arguments. Some accept a fixed number of required arguments plus some number of optional arguments. We will follow the convention used in the official Python documentation, which encloses optional arguments in square brackets. Some functions can even take an arbitrary number of arguments, which is shown by the use of an ellipsis.

Python has a fairly small number of "built-in" functions. Some of the more frequently used are:

`len(`*`arg`*`)`
: Returns the number of characters in ***`arg`*** (although it's actually more general than that, as will be discussed later)

`print(`*`args`*`...[, sep=`*`seprstr`*`][, end=`*`endstr`*`])`
: Prints the arguments, of which there may be any number, separating each by a ***`seprstr`*** (default `' '`) and omitting certain technical details such as the quotes surrounding a string, and ending with an ***`endstr`*** (default `'\n'`)

> **Python 2:** `print` is a statement, not a function. There is no way to specify a separator. The only control over the end is that a final comma suppresses the newline.

`input(`*`string`*`)`
: Prompts the user by printing ***`string`***, reads a line of input typed by the user (which ends when the Return or Enter key is pressed), and returns the line as a string

> **Python 2:** The function's name is `raw_input`.

Here are a few examples:

```
>>> len('TATA')
4
>>> print('AAT', 'AAC', 'AAG', 'AAA')
AAT AAC AAG AAA
>>> input('Enter a codon: ')
Enter a codon: CGC
'CGC'
>>>
```

Here are some common numeric functions in Python:

`abs(`*value*`)`
: Returns the absolute value of its argument

`max(`*args...*`)`
: Returns the maximum value of its arguments

`min(`*args...*`)`
: Returns the minimum value of its arguments

Types can be called as functions too. They take an argument and return a value of the type called. For example:

`str(`*arg*`)`
: Returns a string representation of its argument

`int(`*arg*`)`
: Returns an integer derived from its argument

`float(`*arg*`)`
: Returns a float derived from its argument

`bool(`*arg*`)`
: Returns `False` for `None`, zeros, empty strings, etc., and `True` otherwise; rarely used, because other types of values are automatically converted to Boolean values wherever Boolean values are expected

Here are some examples of these functions in action:

```
>>> str(len('TATA'))
'4'
>>> int(2.1)
2
>>> int('44')
44
>>> bool('')
False
>>> bool(' ')
True
>>> float(3)
3.0
```

> Using `int` is the only way to guarantee that the result of a division is an integer. As noted earlier, `//` is the floor operator and results in a float if either operand is a float.

There is a built-in help facility for use in the Python interpreter. Until we've explored more of Python, much of what the help functions print will probably appear strange or even unintelligible. Nevertheless, the help facility is a useful tool even at this early stage. You can use either of these commands to access it:

`help()`
: Enters the interactive help facility

`help(x)`
: Prints information about `x`, which can be anything (a value, a type, a function, etc.); help for a type generally includes a long list of things that are part of the type's implementation but not its general use, indicated by names beginning with underscores

Occasionally your code needs to test whether a value is an instance of a certain type; for example, it may do one thing with strings and another with numbers. You can do this with the following built-in function:

`isinstance(x, sometype)`
: Returns `True` if `x` is an instance of the type (class) ***`sometype`***, and `False` otherwise

Method calls

Many different types of values can be supplied as arguments to Python's built-in functions. Most functions, however, are part of the implementation of a specific type. These are called *methods*. Calling a method is just like calling a function, except that the first argument goes before the function name, followed by a period. For example, the method `count` returns the number of times its argument appears in the string that precedes it in the call. The following example returns `2` because the string `'DL'` appears twice in the longer string:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'.count('DL')
2
```

Except for having their first argument before the function name, calls to methods have the same features as calls to ordinary functions: optional arguments, indefinite number of arguments, etc. Here are some commonly used methods of the `str` type:

`string1``.count(`*`string2`*`[, `*`start`*`[, `*`end`*`]])`
: Returns the number of times ***`string2`*** appears in ***`string1`***. If ***`start`*** is specified, starts counting at that position in ***`string1`***; if ***`end`*** is also specified, stops counting before that position in ***`string1`***.

`string1``.find(`*`string2`*`[, `*`start`*`[, `*`end`*`]])`
: Returns the position of the last occurrence of ***`string2`*** in ***`string1`***; `-1` means ***`string2`*** was not found in ***`string1`***. If ***`start`*** is specified, starts searching at that position in ***`string1`***; if ***`end`*** is also specified, stops searching before that position in ***`string1`***.

`string1``.startswith(`*`string2`*`[, `*`start`*`[, `*`end`*`]])`
: Returns `True` or `False` according to whether ***`string2`*** starts with ***`string1`***. If ***`start`*** is specified, uses that as the position at which to start the comparison; if ***`end`*** is also specified, stops searching before that position in ***`string1`***.

string1.strip([*string2*])
: Returns a string with all characters in *string2* removed from its beginning and end; if *string2* is not specified, all whitespace is removed.

string1.lstrip([*string2*])
: Returns a string with all characters in *string2* removed from its beginning; if *string2* is not specified, all whitespace is removed.

string1.rstrip([*string2*])
: Returns a string with all characters in *string2* removed from its end; if *string2* is not specified, all whitespace is removed.

Here are some examples of method calls in action:

```
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'.find('DL')
4
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'.find('DL', 5)
14
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'.find('DL', 5, 12)
-1
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'.startswith('DL')
False
>>> 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'.startswith('DL', 4)
True
```

The restriction enzyme with the amino acid sequence in these examples recognizes the site with the base sequence TCCGGA. It's easy enough to find the first location in a DNA base sequence where this occurs:

```
>>> 'AAAAATCCCGAGGCGGCTATATAGGGCTCCGGAGGCGTAATATAAAA'.find('TCCGGA')
27
>>>
```

If the recognition site did not occur in the sequence, `find` would have returned `-1`.

Compound Expressions

The examples of operator expressions that we've looked at thus far have had only a single operator. However, just as in traditional algebra, operators can be compounded in a series. For example:

```
>>> 2 * 3 + 4 - 1
9
```

This is read as "2*3 is 6, 6+4 is 10, and 10–1 is 9." The story isn't quite that simple, though. Consider the following example:

```
>>> 4 + 2 * 3 - 1
9
```

Reading from left to right, we'd have "4+2 is 6, 6*3 is 18, 18–1 is 17," not 9. So why do we get 9 as the result? Programming languages incorporate *operator precedence rules* that determine the order in which operations in a series should be performed. Like

most programming languages, Python performs multiplications and divisions first and then goes back and performs additions and subtractions.

You can indicate your intended interpretation of a sequence of operations by surrounding parts of an expression with parentheses. Everything inside a pair of parentheses will be evaluated completely before the result is used in another operation. For instance, parentheses could be used as follows to make the result of the preceding example be 17:

```
>>> (4 + 2) * 3 - 1
17
```

Comparisons can be combined to form "between" expressions:

```
>>> 1 < 4 < 6
True
>>> 2 <= 2 < 5
True
>>> 2 < 2 < 5
False
```

Strings can participate in sequences of operations:

```
>>> 'tc' in ('ttt' + 'ccc' + 'ggg' + 'aaa')
True
>>> 'tc' in 't' * 3 + 'c' * 3 + 'g' * 3 + 'a' * 3
True
```

The second variation demonstrates that * has a higher precedence than `+`, and `+` has a higher precedence than `in`. Don't hesitate to use parentheses if you have any doubt about the interpretation of operation series.

Here is a list of the operators mentioned in this chapter, ordered from highest precedence to lowest:

- Calls
- Slicings
- Subscriptions
- Exponentiation (**)
- Unary `+`, `-`
- Multiplication, division, and remainder (`*`, `/`, `//`, `%`)
- Addition and subtraction (`+`, `-`)
- Comparisons (`==`, `!=`, `<`, `<=`, `>`, `>=`)
- Membership (`in`, `not in`)
- Boolean not (`not`)
- Boolean and (`and`)
- Boolean or (`or`)

Tips, Traps, and Tracebacks

Tips

- *Don't trust what you see!* Everything printed out in a computing environment or by a programming language is an interpretation of an internal representation. The visible interpretation may not be what you anticipated, even though the internal representation is actually the result you expected.

Statements and expressions

- The results of `and` and `or` expressions are not converted to Booleans. For `and` expressions, the first operand is returned if it is false, and otherwise the second operand is returned. For `or` expressions, the first operand is returned if it is true, and otherwise the second operand is returned. For example, `'' and 'A'` evaluates to `''`, not `False`, while `'' or 'A'` evaluates to `'A'`, not `True`. We'll see examples later of idioms based on this behavior.
- Function calls are both expressions and statements.
- Experiment with using the `sep` and `end` keyword arguments to `print`. They give you more control over your output. The default is to separate every argument by a space and end with a newline.
- A method call is simply a function call with its first argument moved before the function name, followed by a period.
- If you are ever in doubt about the order in which an expression's operations are performed, use parentheses to indicate the ordering you want. Parentheses can sometimes help make the code more readable. They are never required in operation expressions.

Running Python interactively

- Start the Python interpreter from the command line[§] by typing `python` at a command prompt. Here are a few points to keep in mind:
  - —If the initial message you see when Python starts indicates that its version number begins with a 2, exit and try typing `python3`. If that doesn't work, try including a version number (e.g., `python3.1` or `python3.2`).
  - —If *that* doesn't work, either you don't have Python 3 installed or it's not on the path used in your command-line environment. If you don't know how to add

§ *Command line* is a term that refers to an interactive terminal-like window: a Unix shell, OS X Terminal window, or Windows Command window. The command line prompts for input and executes the commands you type.

it, find someone knowledgeable about the command-line environment in your operating system to help you resolve the problem.

- The way to exit Python follows each platform's usual conventions: Ctrl-D on Unix-based systems, Ctrl-Z on Windows variants. You can also type `quit()`.
- In Unix and OS X shells, depending on how Python was installed, you may be able to edit the current line you are typing to Python and navigate back and forth in the history of inputs.‖ After you've typed at least one line to Python, try Ctrl-P or the up arrow. If that changes the input to what you typed previously, the editing capability is functioning. You can use Ctrl-P or the down arrow to move to a later line in the input history. Following are some editing operations that work on the current line:

  Ctrl-A
  : Go to the beginning of the line.

  Ctrl-E
  : Go to the end of the line.

  Ctrl-B or left arrow
  : Move one character to the left.

  Ctrl-F or right arrow
  : Move one character to the right.

  Backspace
  : Delete the preceding character.

  Ctrl-D
  : Delete the next character.

  Ctrl-K
  : Delete the rest of the line after the cursor.

  Ctrl-Y
  : "Yank" the last killed text into the line at the location of the cursor.

  Ctrl-_ (underscore)
  : Undo; can be repeated.

  Ctrl-R
  : Search incrementally for a preceding input line.

  Ctrl-S
  : Search incrementally for a subsequent input line.

‖ If not, the Python you are using was built without the `readline` system (not Python) library. If you configured, compiled, and installed Python yourself, you probably know how to get the `readline` library, install it, and repeat the configure-compile-install process. If not, you will have no idea what any of this is about, and there probably isn't anything you can do about it.

Return
: Give the current line to the interpreter. Similar functionality may be available when Python is run in a Windows command window.

Traps

- The value of a floor division (`//`) *equals* an integer but has the *type* `int` only if both operands were `int`s; otherwise, the value is a `float` that prints with a `0` after the decimal point.
- The result of an operation with a `float` operand may produce a result very slightly more or very slightly less than its "true" mathematical equivalent.
- Remember that the first element of a string is at index `0` and the last at `-1`.
- The index in a string indexing expression must be greater than or equal to `0` and less than the length of the string. (The restriction does not apply to slices.)
- In a function call with more than one argument, every argument except the last must be followed by a comma. Usually omitting a comma will cause syntax errors, but in some situations you will accidentally end up with a syntactically correct expression that is not what you intended.
- Omitting a right parenthesis that closes a function call's argument list results in a syntax error message pointing to the line *after* the one containing the function call.
- Function and method calls with no arguments must still be followed by (an empty pair of) parentheses. Failing to include them will *not* lead to a syntax error, because the value of the name of the function is the function itself—a legitimate value—but it will lead to very unexpected results and, often, runtime errors.

Tracebacks

Representative error messages include:

`NameError: 'Non' is not defined`
: Python doesn't recognize a name (more on this in the next chapter).

`IndexError: string index out of range`
: For a string of length `N`, an index (i.e., the value between square brackets) must be in the range `-N <= index < N-1`.

`SyntaxError`
: Python syntax violation.

`ZeroDivisionError`
: `/`, `//`, or `%` with `0` as the second operand.

CHAPTER 2
Names, Functions, and Modules

In this chapter we'll see how to name values, define new functions, and incorporate optional software from the Python library. All of these operations rest on Python mechanisms for naming and interpreting names.

A Python *name* consists of an arbitrary number of letters, underscores, and digits. The only restriction is that the first character must not be a digit (otherwise Python would interpret the digit as the beginning of a number). Names beginning with two underscores are special in one of several ways that will be explained later, so names you choose will almost always begin with a letter.

A name is used to refer to something—a primitive value, a function, or any of a number of other possibilities. *A name is not the same as a string*. A name refers to something, whereas a string has no intrinsic meaning to Python. Unlike strings, names are not enclosed in quotes. Giving a name to a value is called *binding*. One value may have multiple names bound to it, as Figure 2-1 illustrates.

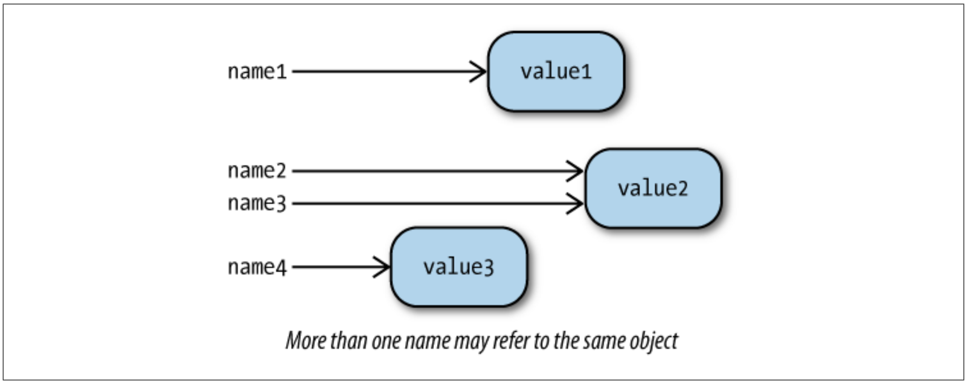


Figure 2-1. Names bound to objects

The same name can mean different things in different contexts. The technical term for such a context is a *namespace*. The types and built-in functions we saw in the previous

chapter are all part of the *global* namespace. A separate namespace is associated with each type, such as `str`. This enables each type to have its own version of common methods, such as `count` and `find`. Which version gets executed is determined by the type of the value through which the method is called. We'll see more details about all of this later, but Figure 2-2 should give you an idea of how namespaces in Python work.

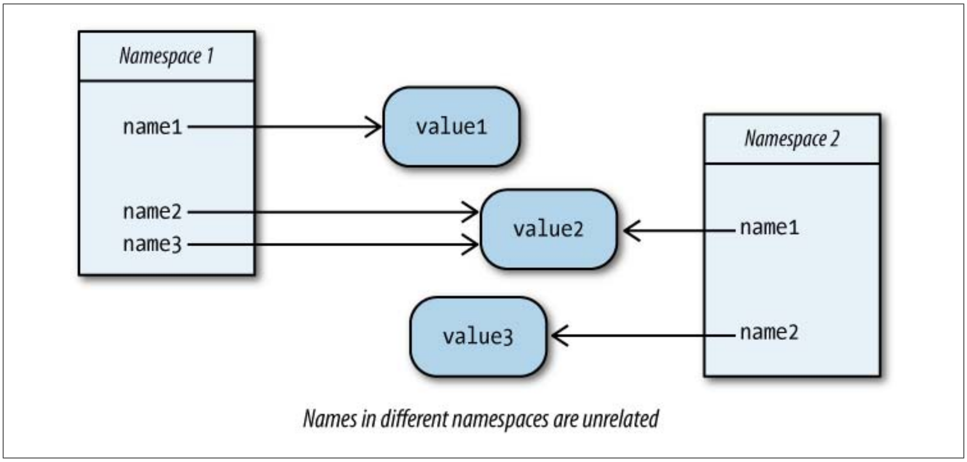


Figure 2-2. Names in different namespaces bound to objects

We've already seen some kinds of things that have names: types such as `str`, functions such as `len`, and methods such as `str.count`. Here's what Python responds with when given these names as input:

```
>>> str
<class 'str'>
>>> len
<built-in function len>
>>> str.count
<method 'count' of 'str' objects>
```

In Python, *class* is a synonym for "type" and *object* is a synonym for "value."* The period in `str.count` tells Python to look for the named method in the class `str`. `'actg'.count` has a similar effect, because the type of `'actg'` is `str`. The statement "every value has a type" can be rephrased as "every object is an instance of a class." Classes and instances form the basis of *object-oriented programming*. Built-in types and types loaded from Python libraries take advantage of Python's object-oriented features. Chapter 5 discusses how to use those to build your own classes, but until then all we'll need is the little bit of object-oriented vocabulary just introduced.

* Older versions of Python did distinguish classes from types and objects from values.

Assigning Names

A Python program consists of a series of *statements*. Statements do not produce values and cannot be used as expressions.

STATEMENT

Assignment

An *assignment statement* binds a name to an object. Assignment is denoted by a single equals sign:

```
name = value
```

All an assignment statement does is name a value. That value can also have other names. Assigning a name to a value never changes that value, or any of its other names. Here are some examples of assignment statements and their effects:

```
>>> aaseq1 = 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'     # no value printed
>>> aaseq2 =  'AARHQGRGAPCGESFWHWALGADGGHGHAQPPFRSSRLIGAERQPTSDCRQSLQ'
>>> aaseq1
'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'
>>> len(aaseq2)
54
>>> aaseq3                                         # never bound
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    aaseq3
NameError: name 'aaseq3' is not defined
>>> aaseq3 = aaseq1
>>> aaseq3
'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'
```

Assignment statements may be chained together. This feature is used primarily to bind several names to some initial value:

```
>>> a = b = c = 0
```

Another variation on assignment statements is a bit of shorthand for arithmetic updates. Statements like the following are so common:

```
>>> a = a + 1
```

that Python provides a shorthand for them:

```
>>> a += 1
```

This is called an *augmented assignment statement*. The operation of an augmented assignment statement can be any of the arithmetic operations shown in the previous chapter: +=, -=, *=, /=, //=, %=, and **=.

Defining Functions

New functions are defined with *function definition statements*. A definition is a *compound statement*, meaning it comprises more than one line of code. We will see other kinds of compound statements later, especially in Chapter 4. The first line of each compound statement ends with a colon. Subsequent lines are indented relative to the first.

The standard practice is to indent by four spaces, and you should adhere to that rule. Also, do not use tab characters since they confuse Python's indentation computations.

STATEMENT

Function Definition

Definition statements have the general form:

```
def name(parameter-list):
    body
```

where ***body*** is one or more statements, each indented relative to the line with the `def`. The ***parameter-list*** is a (possibly empty) list of names. If there is more than one parameter in the list, they are separated by commas.

When one function calls another, Python records the call from the first function and passes control to the second. If that one calls a third, its call would be recorded and control passed to the third. The record of a call includes the location of the function call in the caller's code so that when a function regains control it can continue executing the code following the function call. Figure 2-3 illustrates how function calling works, with the curved arrows representing the call records.

Some functions are intended to return a value and some aren't. A `return` statement is used to return a value from a function. The word `return` is usually followed by an expression whose value is returned as the value of the function. Occasionally a `return` is not followed by an expression, in which case the value returned is `None`. All functions return a value, whether or not they contain a `return` statement: if the function finishes executing without encountering a `return`, it returns `None`.

STATEMENT

Function Return

`return` exits the currently executing function, returning ***value*** to its caller. ***value*** may be omitted, in which case `None` is returned.

```
return value
```

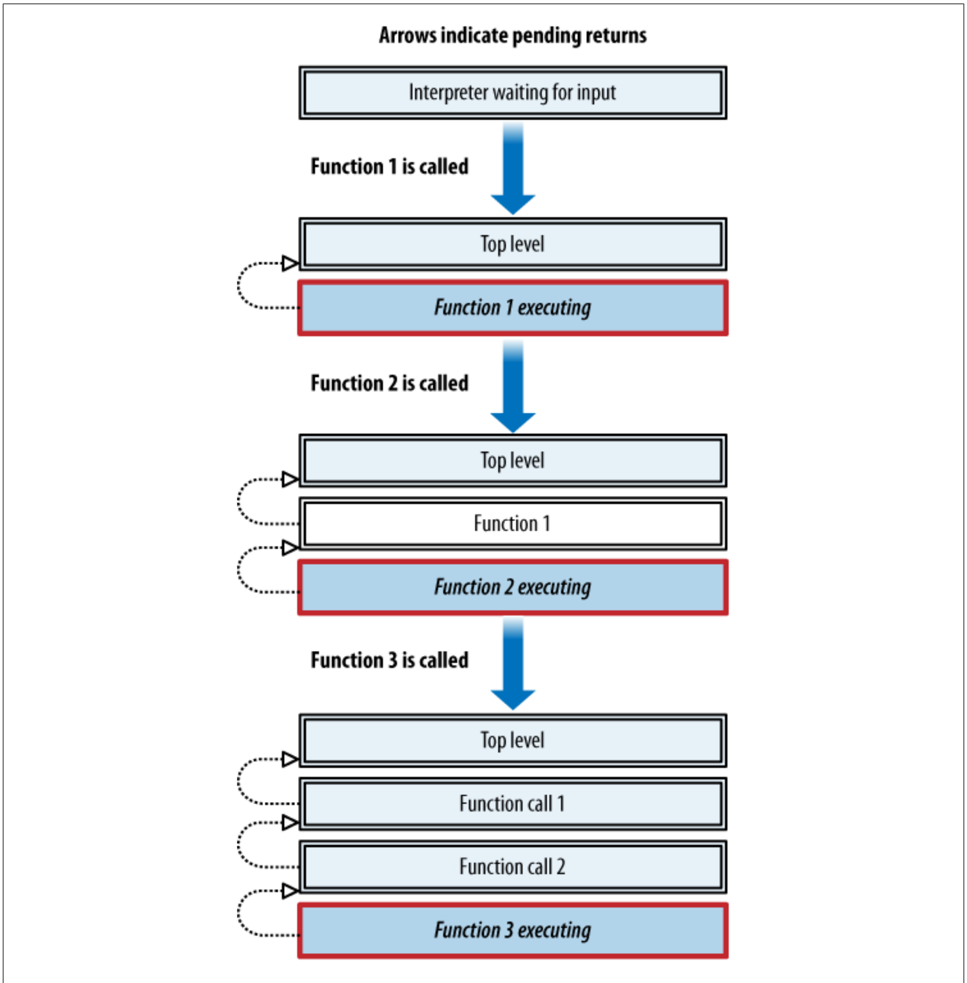


Figure 2-3. Function calls

If you call a function from the interpreter or an IDE while you are developing code and nothing is printed, that means your function returned `None`. *A very common reason for this is that you simply forgot to include a* `return` *statement in the function definition*. Even experienced Python programmers stumble over this from time to time.

When a function returns, the function that called it continues executing at the point in its code following the call. This happens whether the function returns by executing all its statements or by executing a `return` statement. Once the function returns, Python no longer needs the information associated with its call. Figure 2-4 illustrates the returns from the functions shown in Figure 2-3.

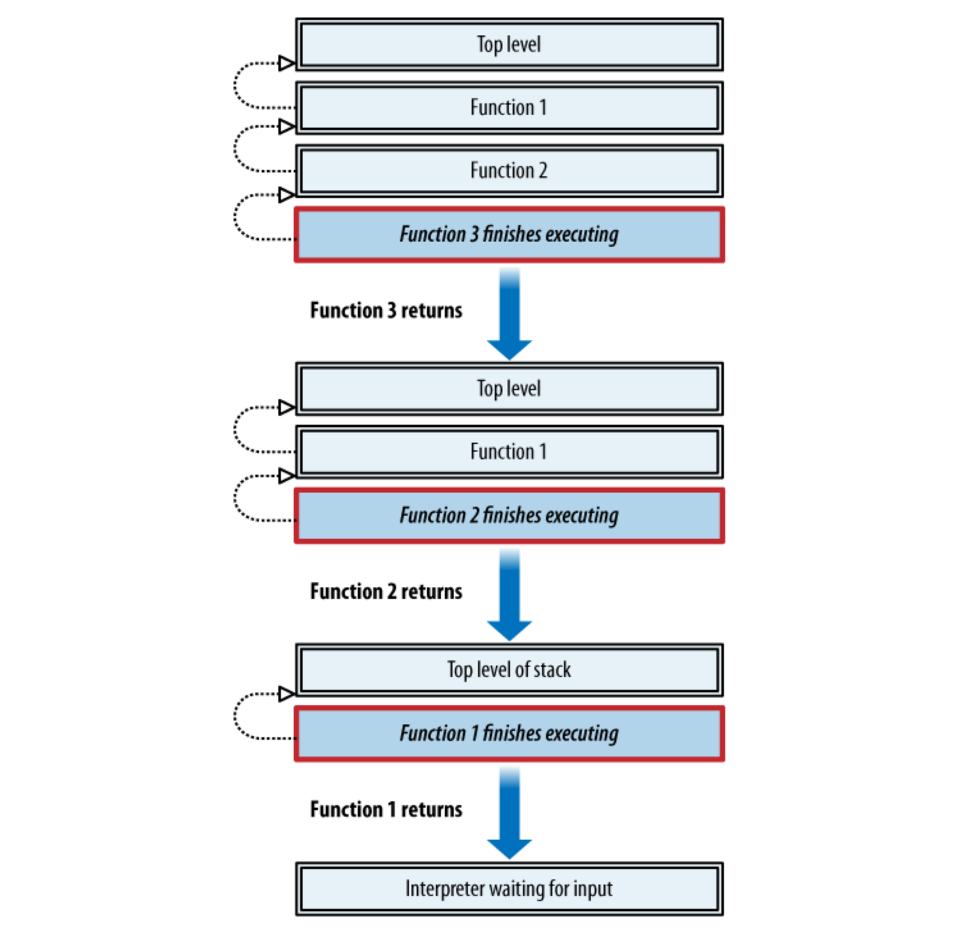


Figure 2-4. Function returns

When you start working on a function, you might not know what to put in the body until you've done some more experimentation. A function's body must contain at least one statement. The do-nothing statement that is equivalent to the no-value `None` is the simple statement `pass`.

STATEMENT

Do Nothing

The `pass` statement does nothing. It is used as a placeholder in compound statements when you haven't yet decided what to actually put there.

```
pass
```

A minimal function definition would therefore be:

```
def fn():
    pass
```

Now `fn` can be called, though it does nothing and returns nothing:

```
>>> fn()
>>>
```

Words such as `def`, `return`, and `pass` are known as *keywords* or, because programs cannot use them as names, *reserved words*. `True`, `False`, and `None` are keywords, as are operators such as `and`, `or`, `not`, and `in`. A complete list of keywords is found in Table A-1 in Appendix A.

Function Parameters

When a function is called, Python evaluates each of the call's argument expressions. Then it assigns the first parameter name to the first argument's value, the second parameter name to the second value, and so on, for however many parameters there are. Once all the parameter names have been assigned, the statements in the function body are executed. Example 2-1 shows a definition based on a simple use of `str.find`.

Example 2-1. A simple function for recognizing a binding site

```
def recognition_site(base_seq, recognition_seq):
    return base_seq.find(recognition_seq)
```

Each function has a separate namespace associated with it. The function's parameters go in that namespace. Assignment statements in the function body also create bindings within the function's namespace. Therefore, a name bound in the interpreter and a name with the same spelling bound in a function definition are in different namespaces and have nothing to do with each other. Example 2-2 illustrates these details.

Example 2-2. Some function definition details

```
def validate_base_sequence(base_sequence): ❶
    seq = base_sequence.upper() ❷
    return len(seq) == (seq.count('T') + seq.count('C') + ❸
                        seq.count('A') + seq.count('G')) ❹

>>> seq = 'AAAT'
>>> validate_base_sequence('tattattat')
True
>>> seq
'AAAT'
```

Line ❶ begins the definition of a function called `validate_base_sequence` that has one parameter, `base_sequence`. Line ❷ assigns the name `seq` to the result of calling the method `upper` on the value of `base_sequence`. The purpose of line ❷ is to simplify the rest of the definition by ensuring that all the base characters will be uppercase even if

the function is called with a lowercase string. By assigning a name to the result of `upper`, we avoid having to repeat that call four times later.

Lines ❸ and ❹ compare the length of the sequence string to the sum of the number of Ts, Cs, As, and Gs in the sequence string. If the length is equal to that sum, the function returns `True`; otherwise, there is a character in the parameter sequence that isn't a valid base character, and the result will be `False`.

Note that the right side of the equality expression beginning on line ❸ continues on line ❹. This is because it begins with a left parenthesis, and Python won't consider it complete until it sees a matching right parenthesis. Without the opening parenthesis, the interpreter would produce an error message as soon as it encountered the end of line ❷. Line breaks can appear within a parenthesized expression, giving you more control over how you format long expressions.

Another way to continue a long line without using parentheses is to end the line with a backslash; this tells Python that the current line is continued on the next. (Usually it's preferable to rely on parentheses for multiline expressions, but occasionally backslashes are a better choice.) Example 2-3 shows Example 2-2 rewritten using a backslash instead of the outermost parentheses.

Example 2-3. Using backslash for line continuations

```
def validate_base_sequence(base_sequence):
    seq = base_sequence.upper()
    return len(seq) == \
                seq.count('A') + seq.count('G') + \
                seq.count('T') + seq.count('C')
```

The definition is followed by a blank line. What is significant is not that the line is blank but that it is not indented to a position beyond that of the `def` that begins the definition. A block in Python ends with the first line that is indented less than the first line of the block—i.e., the line following the one that ends with a colon.

Comments and Documentation

The `#` character tells Python to ignore the rest of the line. This convention can be used for a short comment following the code on a line, or for a more extensive comment consisting of one or more lines beginning with a `#`. For instance, we could add comments to the `validate_base_sequence` definition, as in Example 2-4.

Example 2-4. Documentation of a function with commented lines

```
# A short example illustrating various details of function definitions. Given a string ostensibly
# representing a base sequence, return True or False according to whether the string is composed
# entirely of upper- or lowercase T, C, A, and G characters
def validate_base_sequence(base_sequence):
    # argument should be a string
    seq = base_sequence.upper()
    # ensure all uppercase characters
```

```
    return len(seq) == (seq.count('T') + seq.count('C') +
                        seq.count('A') + seq.count('G'))
```

In general, the kind of information provided in the comment lines before the function definition in this example should be inside the definition, not in a comment before it. Python has a separate feature for this called a *docstring*. Syntactically, it is simply a string that begins the function definition. (Because it is often more than one line long, triple quotes—single or double—are normally used.) Example 2-5 shows the comment of Example 2-4 changed to a docstring.

Example 2-5. A function definition with a docstring

```
def validate_base_sequence(base_sequence):
    """Return True if the string base_sequence contains only
    upper- or lowercase T, C, A, and G characters, otherwise False"""
    seq = base_seq.upper()
    return len(seq) == (seq.count('T') + seq.count('C') +
                        seq.count('A') + seq.count('G'))
```

Normally, the only kinds of expressions that have any effect when used as statements are function and method calls; docstrings are an exception. Docstrings are different from comments. Comments disappear when Python interprets code, but docstrings are retained. Consequently, docstrings have greater utility than comments. In particular, the `help` function looks at the docstring of a user-defined function together with its parameter list to generate a help description, as shown in Example 2-6. (The phrase `in module __main__` simply means "defined in the interpreter.")

Example 2-6. Help called on a user-defined function

```
>>> help(validate_base_sequence)
Help on function validate_base_sequence in module __main__:

validate_base_sequence(base_sequence)
    Return True if the string base_sequence contains only
    upper- or lowercase T, C, A, and G characters, otherwise False
```

> Function definitions contain only statements. A string by itself is a statement because it is a value, a value is an expression, and an expression is a statement. However, docstrings are the only place where a freestanding string would have any effect.

Let's look at another example. Genomes of different species—and regions within a genome—vary with respect to the proportion of Gs and Cs in their DNA as opposed to Ts and As. It is a straightforward exercise to compute the *GC content* of a given DNA sequence represented as a string. One of the advantages of an interactive environment such as Python's interpreter is that you can experiment with small pieces of code before putting them together. Here's what such experimentation might look like in working toward a definition of a `gc_content` function:

```
>>> seq = 'ATCCGGGG'
>>> seq.count('G')
4
>>> seq.count('C')
2
>>> len(seq)
8
>>> seq.count('G') + seq.count('C')
6
>>> (seq.count('G') + seq.count('C')) / len(seq)
0.75
```

The end result is shown in Example 2-7.

Example 2-7. Defining a function to compute GC content

```
def gc_content(base_seq):
    """Return the percentage of G and C characters in base_seq"""
    seq = base_seq.upper()
    return (seq.count('G') + seq.count('C')) / len(seq)
```

A meaningful example of the use of this function would require a long base sequence, and then it would be difficult to tell whether the result was correct. When writing functions, it is generally a good idea to construct some small "test cases" for which the correct answer is known:

```
>>> seq50 = 'AACCTTGG'
>>> seq75 = 'ATCCCGGG'
>>> seq40 = 'ATATTTCGCG'
>>> gc_content(seq50)
0.5
>>> gc_content(seq75)
0.75
>>> gc_content(seq40)
0.4
```

Assertions

While developing code, it often happens that your functions get called with arguments of the wrong type or value. Depending on the details of the code, an invalid argument value may result in a Python error or a nonsensical result. Even worse, it might result in an apparently meaningful, but incorrect, result. For example, the definition of `gc_content` assumes that it gets called with a string that contains only (upper- or lowercase) Ts, Cs, As, and Gs. Since it is only counting Cs and Gs, all other characters are assumed to be Ts or As. If given a string that contains a few invalid characters, it will return a value that "makes sense" but is nonetheless "off" by a small amount.

It's often a good idea to express a function's assumptions in its docstring. However, the documentation doesn't have any effect on the computation—nothing stops someone from calling a function with values that violate the documented assumptions. To

ensure compliance Python provides a simple *assertion statement*, one of whose uses is to validate arguments.

> **STATEMENT**
>
> ## Assertion
>
> An *assertion statement* tests whether an expression is true or false, causing an error if it is false.
>
> **assert** *expression*

Example 2-8 illustrates what happens when an assertion fails.

Example 2-8. A failed assertion

```
>>> assert 1 == 2
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    assert 1 == 2
AssertionError
```

There is another form of the assertion statement that takes two expressions. If the first expression is false, the second becomes part of the error report.

> **STATEMENT**
>
> ## Two-Expression Assertion
>
> A two-expression assertion statement takes two arguments: an expression to evaluate and an expression to use in the error report if the first expression is false.
>
> **assert** *expression1*, *expression2*

Example 2-9 shows a two-expression assertion.

Example 2-9. A failed assertion with a second expression

```
>>> assert 1 == 2, 'invalid arguments'
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    assert 1 == 2, "invalid arguments"
AssertionError: invalid argument
```

Let's improve `gc_content` by ensuring that its argument is a valid sequence string. We already have a function that does what we need—`validate_base_sequence`. We'll add an assertion statement that calls that function. The assertion in Example 2-10 both documents the function's assumptions and enforces them when the function is called.

Example 2-10. Adding an assertion to the gc_content function

```
def gc_content(base_seq):
    """Return the percentage of G and C characters in base_seq"""
    assert validate_base_sequence(base_seq), \
           'argument has invalid characters'
    seq = base_seq.upper()
    return ((base_seq.count('G') + base_seq.count('C')) /
            len(base_seq))
```

The function in Example 2-10 calls another function that we've defined. This is quite common. Each function should perform what is conceptually a single coherent action. This means you'll write many relatively small functions that call each other rather than writing large functions with many steps.

You can think of the functions you define as building up a *vocabulary*. In defining a vocabulary, it is natural to define some of its words in terms of others. Python provides a small initial set of words, and you expand on that set by defining more of them.

One reason for writing small functions is that simple functions are easier to write and test than complicated ones. If function A calls B, and B calls C, and C calls D, but something isn't working in D, you can call D yourself from the interpreter. Once you're sure that D works, you can call C, and so on. Another reason for defining a separate function for each action is that a very focused function is likely to prove more useful in other definitions than a larger multipurpose one.

Writing many small functions helps avoid code duplication, that is, having a group of statements repeated in more than one definition. Code duplication is bad because if you want to change those statements you'll have to find every definition that uses them and spend considerable time making the same changes in each definition. Changing multiple occurrences risks ending up with inconsistencies.

Default Parameter Values

You'll often find that most calls to a certain function are likely to include the same value for a particular parameter. Frequently this value is something simple such as `True`, `0`, or `None`. In such cases, Python provides a way to assign a *default value* to the parameter that will be used if no explicit value is included in a call to the function.

As mentioned in the preceding chapter, some parameters of some functions are optional. A function definition designates an optional parameter by assigning it a default value in the parameter list. There can be any number of parameters with default values. All parameters with default values must follow any parameters that do not have them.

By way of example, we'll make `validate_base_sequence` more flexible by giving it the ability to handle RNA sequences too. This simply entails checking for a U instead of a T. Example 2-11 adds a second parameter whose value determines whether the function looks for Ts or Us. The term or syllable *flag* (sometimes abbreviated to `flg`) is commonly used for a Boolean value that determines which of two actions is performed. We'll call

our new parameter RNAflag. By using a three-part conditional expression, we can make the change with very little rewriting.

Example 2-11. Adding a "flag" parameter

```
def validate_base_sequence(base_sequence, RNAflag):
    """Return True if the string base_sequence contains only upper- or lowercase
    T (or U, if RNAflag), C, A, and G characters, otherwise False"""
    seq = base_sequence.upper()
    return len(seq) == (seq.count('U' if RNAflag else 'T') +
                        seq.count('C') +
                        seq.count('A') +
                        seq.count('G'))

>>> validate_base_sequence('ATCG', False)
True
>>> validate_base_sequence('ATCG', True)
False
>>> validate_base_sequence('AUCG', True)
True
```

The new definition requires calls to the function to provide two arguments. To make using the function a bit more convenient, we can give RNAFlag a default value, as shown in Example 2-12. We don't really know whether the function will get used more often for DNA sequences or RNA sequences, so it's hard to choose a default value on that basis. In general, default values should be chosen to "turn off" the feature a parameter provides, so that users of the function who don't know about the feature aren't surprised by its use. A common default is the kind of false value appropriate for the situation: False, an empty string, a zero, etc. Flags, being Boolean, normally have default values of False.

Example 2-12. Adding a default value for the flag parameter

```
def validate_base_sequence(base_sequence, RNAflag=False):
    """Return True if the string base_sequence contains only upper- or lowercase
    T (or U, if RNAflag), C, A, and G characters, otherwise False"""
    seq = base_sequence.upper()
    return len(seq) == (seq.count('U' if RNAflag else 'T') +
                        seq.count('C') +
                        seq.count('A') +
                        seq.count('G'))

>>> validate_base_sequence('ATCG', False)
True
>>> validate_base_sequence('ATCG', True)
False
>>> validate_base_sequence('AUCG', True)
True
>>> validate_base_sequence('AUCG')
>>> # second argument omitted; defaults to True
True
```

Keyword parameters have the following properties:

- They are optional (i.e., do not need to be included in calls to the function).
- They are defined by specifying default values in the function's parameter list; if the keyword does not appear in a call to the function, its default value is used.
- In both the parameter list of a function definition and the argument list of its calls, all "positional" (i.e., required) arguments must appear before any keyword parameters.
- Keyword parameters may appear in any order.
- With a few exceptions, keyword arguments may be provided positionally in a function call in the order in which the keywords appear in the function's definition.

Using Modules

In addition to those built into the base interpreter environment, Python offers a large selection of optional types, functions, and methods. These are defined by *module* files placed in a *library* directory as part of Python's installation. Modules can also be obtained from external sources and added to Python's library.

A module file is an ordinary file containing Python statements (`def`s, mostly). A docstring at the beginning of the file (if present) describes its contents and purpose. Later chapters of this book discuss many modules important for bioinformatics programming. Here, I'll only explain the mechanics of module use and give a few basic examples.

Importing

A module's contents are brought into the interpreter's environment by an *import statement*.

STATEMENT

Importing

The basic form of the `import` statement loads a module into the Python environment and makes its name available in the namespace into which it was imported (usually the interpreter or a program).

```
import name
```

The *name* is just the name of the module—no path and no extension.

For example, the module `os` provides an interface to the computer's operating system. You can import it using this statement:

```
>>> import os
```

Python keeps track of the directories in which modules may be found and of which modules have already been imported. The first time a module is imported—with any

form of `import` statement—Python looks for it in each library directory in turn until it is found. It then executes the file's statements and creates an object to represent the module and its contents. Subsequent imports skip all of this.

Python incorporates quite a few modules during its build process. While their *contents* are in the system, their *names* are not. Consider the `os` module just mentioned—all the contents of the module are actually already in Python, but the name `os` has no meaning until the module is imported. Importing assigns the name of the module to the object that represents the module's information.

Module namespaces

Each module has a separate namespace associated with it. In fact, a module object is little more than a namespace. Module contents are accessed using the same dot notation used for method calls. Some modules even have submodules, which are referenced using dot notation—for example, `os.path`. Let's look at some examples:

```
>>> os                                          # after import
<module 'os' from '/usr/lib/python31/os.py'>
>>> os.getcwd()                                 # a function in the os module
'/Users/mlm/programming/python'
>>> os.getlogin()                               # another function in the os module
'mlm'
```

Each module's namespace is isolated from the interpreter's namespace and the namespaces of other modules. Functions with the same name can be defined in multiple namespaces and not interfere with each other, because each is accessed by a name in a different namespace.

Another form of the `import` statement is used to purposely bring a name from a module into the namespace in which the import appears—the interpreter or another module.

STATEMENT

Selective Import

This form of `import` statement loads a library into the Python environment but does not make its name available in the namespace from which it was loaded. Instead, it imports specific names from the module into the importing namespace.

```
from modulename import name1, name2, ...
```

You can give something you are importing a different name when you import it by using the following form of the `import` statement:

```
from modulename import actualname as yourname
```

You can import *all* the names from a module with the following variation:

```
from modulename import *
```

In general, it is best to avoid the "import all" form (occasionally a module's documentation or examples suggest you begin a program by importing all of its names, in which case it's probably all right to do so). The reason to avoid this form is that you don't know what names you'll be importing. Some might rebind existing names—this is called a *name conflict*—and lead to strange problems. Even importing specific names using the `from` form isn't necessarily the best idea—when someone else reads your code, or you read it later, using the module prefix makes it clear which names come from which modules and, if there is no prefix, are defined in the file itself.

If your code frequently uses a particular name from a module, you'll find it convenient to import that name into the code's namespace. For example, `sys` is another module that is already in the base Python system without its name being in the interpreter's namespace. One name `sys` provides is `version`, whose value is a string describing Python's version. After executing the following statement:

```
from sys import version
```

your code will be able to refer directly to `version` instead of `sys.version`. Consider the interactions in Example 2-13.

Example 2-13. Importing and namespaces

```
❶ >>> sys
Traceback (most recent call last):
  File "<pyshell#103>", line 1, in <module>
    sys
NameError: name 'sys' is not defined
❷ >>> version
Traceback (most recent call last):
  File "<pyshell#105>", line 1, in <module>
    version
NameError: name 'version' is not defined
❸ >>> from sys import version
❹ >>> version
'3.0 (r30:67503, Jan  2 2009, 12:13:58) \n[GCC 4.0.1 (Apple Inc. build 5488)]'
❺ >>> sys
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in <module>
    sys
NameError: name 'sys' is not defined
❻ >>> import sys
❼ >>> from sys import version_number
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name version_number
❽ >>> import version
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named version
```

This series of inputs demonstrates the following points:

❶ The name `sys` is not in the interpreter's namespace.

❷ The name `version` is not in the interpreter's namespace.

❸ The name `version` is imported from the module `sys`.

❹ The name `version` is now in the interpreter's namespace.

❺ The name `sys` is still not in the interpreter's namespace.

❻ The name `sys` is now in the interpreter's namespace.

❼ The module `sys` does not define anything called `version_number`.

❽ There is no module named `version`.

The random module

One useful module is `random`, which provides various ways to generate random numbers. The function `random.randint` takes two integer arguments and returns a random integer in the range of the first to the second inclusive. For example, `random.randint(1,4)` will return `1`, `2`, `3`, or `4`. (Note that the range is inclusive of the second number, unlike with string slicing, which designates a substring from the first position up to but not including the second position indicated by the slice.)

One good use of `random.randint` is to generate examples and "test cases" to use with code you are developing. Example 2-14 shows how to generate a random codon.

Example 2-14. Generating a random codon

```
from random import randint

def random_base(RNAflag = False):
    return ('UCAG' if RNAflag else 'TCAG')[randint(0,3)]

def random_codon(RNAflag = False):
    return random_base(RNAflag) + random_base(RNAflag) + random_base(RNAflag)
```

Notice how breaking the code into two separate functions, each of which does its own narrowly focused job, helps readability. There's not much reason to avoid using small functions like these. You'll see as you write them that they'll turn out to be useful for other code. It's helpful to think of types, functions, and methods as parts of a vocabulary: types and their instances are nouns, and functions and methods are verbs. Your definitions add to that vocabulary.

Example 2-15 shows a function that simulates single-base mutation. It uses assignment statements to name everything along the way to the final result.

Example 2-15. Naming intermediate results in a function definition

```
from random import randint

def replace_base_randomly_using_names(base_seq):
    """Return a sequence with the base at a randomly selected position of base_seq replaced
    by a base chosen randomly from the three bases that are not at that position"""
    position = randint(0, len(base_seq) - 1)       # -1 because len is one past end

    base = base_seq[position]
    bases = 'TCAG'
    bases.replace(base, '')                          # replace with empty string!
    newbase = bases[randint(0,2)]
    beginning = base_seq[0:position]                 # up to position
    end = base_seq[position+1:]                      # omitting the base at position
    return beginning + newbase + end
```

Example 2-16 shows another version of the same function. This variation uses a complex expression instead of a lot of names. It does use the name `position` to save the value of the call to `randint`, because that is used in two places; if `randint` were called twice, it would return two different values, and the definition needs to use the same value in both places.

Example 2-16. The same function without intermediate result names

```
def replace_base_randomly_using_expression(base_seq):
    position = randint(0, len(base_seq) - 1)
    return (base_seq[0:position] +
            'TCAG'.replace(base_seq[position], '')[randint(0,2)] +
            base_seq[position+1:])
```

There's no clear-cut rule about which style to use. Sometimes naming things improves readability (even taking the place of comments) and aids debugging. Often, however, it doesn't contribute much, especially if the value named is only used once in the definition. In the case of this function, something between these two extremes would probably be best. The compromise version is shown in Example 2-17.

Example 2-17. A function definition with some intermediate names

```
def replace_base_randomly(base_seq):
    position = randint(0, len(base_seq) - 1)
    bases = 'TCAG'.replace(base_seq[position], '')
    return (base_seq[0:position] +
            bases [randint(0,2)] +
            base_seq[position+1:])
```

Python Files

A file of yours that you import becomes a module just as if the file had been in Python's library. Accessing any of its contents requires using the file's name as a prefix, except for specific names imported into the interpreter's namespace using the "from" form of the `import` statement. Normally a file meant for importing assigns names and defines

functions but doesn't do any observable computation when imported. Files intended to be run or executed normally end with function calls and statements that do something observable, such as printing results.

Each Python file must import any modules it uses. That is, if file A uses module M, it must import module M even if it imports another file that imports M or is imported into the interactive interpreter after M has been imported there. Example 2-18 shows a file containing the functions defined in examples earlier in this chapter, along with a new one for testing the others. The test function is called at the end of the file. That call occurs regardless of whether the file is imported or executed.

Example 2-18. A Python file

```
def validate_base_sequence(base_sequence, RNAflag = False):
    """Return True if the string base_sequence contains only upper- or lowercase
    T (or U, if RNAflag), C, A, and G characters, otherwise False"""
    seq = base_sequence.upper()
    return len(seq) == (seq.count('U' if RNAflag else 'T') +
                        seq.count('C') +
                        seq.count('A') +
                        seq.count('G'))

def gc_content(base_seq):
    """Return the percentage of bases in base_seq that are C or G"""
    assert validate_base_sequence(base_seq), \
           'argument has invalid characters'
    seq = base_seq.upper()
    return (base_seq.count('G') +
            base_seq.count('C')) / len(base_seq)

def recognition_site(base_seq, recognition_seq):
   """Return the first position in base_seq where recognition_seq
   occurs, or -1 if not found"""
   return base_seq.find(recognition_seq)

def test():
    assert validate_base_sequence('ACTG')
    assert validate_base_sequence('')
    assert not validate_base_sequence('ACUG')

    assert validate_base_sequence('ACUG', False)
    assert not validate_base_sequence('ACUG', True)
    assert validate_base_sequence('ACTG', True)

    assert .5 == gc_content('ACTG')
    assert 1.0 == gc_content('CCGG')
    assert .25 == gc_content('ACTT')

    print('All tests passed.')

test()
```

Tips, Traps, and Tracebacks

Tips

Names

- One object may have many names.
- Choose clear, descriptive names for functions and arguments.
- Assignment changes what a name means—i.e., which object it references. If a name refers to something that is modifiable (a phenomenon we will encounter in the next chapter), assignment does not modify the object.

Function definitions

- Think of function definitions as adding to Python's vocabulary of built-in functions, types, methods, etc.
- Use `pass` as a placeholder for the body of a function you are defining; it allows you to start the function definition and specify its parameters without it doing anything while you work on the rest of the code. The other code can even call the function at the appropriate place. Later, you can replace the `pass` to define what the function should really do.
- Your function definitions should normally be quite small. Each should do one main thing. Get comfortable with the idea of defining functions that call each other.
- Except perhaps for functions of a couple of lines with self-descriptive names, always include docstrings in your function definitions.
- Read the following (and occasionally reread them, as they will make more sense and seem more helpful the more Python work you do):
  - Style Guide for Python Code
  - Docstring Conventions
  - The Zen of Python, accessed by typing the following at the command line:

    ```
    import this
    ```

> The code examples in this book do *not* adhere to certain aspects of the official docstring conventions. In particular, the conventions call for docstrings longer than one line to begin with a one-line summary and a blank line before continuing with further details, and to end with a blank line followed by the closing quotes on a separate line. Given the large number of examples contained in this book and an approach to writing code that emphasizes very small functions, following the conventions strictly would have resulted in docstrings whose size rivaled that of the code they documented, as well as simply taking up too much space on the page.

Executing code

- *Do not be intimidated by tracebacks*—they identify the nature and location of the problems they are reporting.
- Start putting your code in Python files as soon as possible. Testing bits of code in the interpreter and then copying them into a file is fine, but eventually, as you become more competent in writing Python code, you will want to edit files directly and then run them in the interpreter. You can use whatever text editor you are comfortable with. Save the files in text format with the extension *.py*.
- Learn how to run Python programs from the command line. If you don't know how, ask someone to show you. (They don't have to know Python!) Using the file *echo.py* as an example, and assuming that the `python3` command runs Python:

  1. Type:

     ```
     pythonpath/python3 echo.py
     ```

     where ***pythonpath*** is the full path to the Python executable on your machine.

  2. If the folder containing the Python executable is in your execution path, as defined by system and shell variables, you only need to type:

     ```
     python3 echo.py
     ```

  These commands will execute the code in *echo.py*, then exit. To execute all the code in the file, enter the interpreter and include `-i` (for "interactive") before the name of your file.

- One of the great advantages of using an interactive interpreter is that you can test pieces of your code without having to test entire function definitions or scripts:

  —When you are first defining a function, you can try out bits of code or aspects of Python about which you are uncertain.

  —You can try various ways of expressing a part of a function definition.

  —After assigning names, you can look at their values.

  —You can test functions interactively by calling them with various values.

Using IDLE

The Python installation comes with a simple integrated development environment (IDE) called IDLE. It has a window with the interpreter running in it and it allows you to edit files in other windows. You can start it by double-clicking its icon or by double-clicking a Python file. (You may have to change the default application for *.py* files to open with the Python 3 version of IDLE: on Windows, the default is to run the program in a shell, then exit the shell. On any platform, if a version of Python 2 is present on your computer, you may have to change the default application for *.py* files to be the Python 3 version of IDLE.

When you open a file with IDLE—either by double-clicking it or selecting the Open command from IDLE's File menu—the file will be displayed in an IDLE editor window. Editor windows provide many conveniences, including "syntax highlighting" and keyboard shortcuts. However you open IDLE, you will see a "Python Shell" window that is the equivalent of the command-line interpreter. Here are some hints for using IDLE:

- The typical shortcut keys are available for the Edit menu's commands, for most of the other menu commands, and for some movement and editing commands that are not on a menu. The Preferences command on the IDLE menu opens a tabbed window that you can use to customize IDLE's appearance (including the typeface and font size), shortcut keys, and behavior.
- The IDLE Help command on the Help menu brings up a window explaining the menu commands, keyboard shortcuts for editing and moving around, and other aspects of using IDLE. (This is also available as the file *help.txt* in the *idle* subdirectory of the Python installation's *library* directory.)† *You should periodically review this documentation*, as IDLE has many useful features that you won't at first appreciate.
- When in an editor window, you can execute the code in the window by selecting the Run Module command from the Run menu or pressing F5. (The Run menu is replaced by the Debug menu when the cursor is in the interpreter window.) Every time you execute code using IDLE's Run Module command, the Python interpreter is restarted. No previous assignments, definitions, or imports survive this restart.

  The restart behavior is another reason why it's useful to include everything you need in files you are developing in IDLE, including assignments to strings and numbers that you want to use in testing and debugging your code. Basically, anything you don't want to type repeatedly should go in a file. However, once you've run the code, all of its assignments and definitions will be available in the interpreter until the next time you execute the Run Module command.
- The text in IDLE's editor window is colored to distinguish various aspects of Python code. (The colors are also customizable, so if you find some either too dramatic or too difficult to distinguish, you should take the time to adjust them to your liking.) This feature is called, naturally enough, *syntax highlighting*. It makes the code much more readable. In addition, the color of the text you are presently typing and of the text that follows it can reveal syntax problems without you even running the program.

  Perhaps the most useful example of this is when you notice that the text you are typing is colored as a string, indicating that you did not type the quotes required to end the string where you intended. Incomplete multiline strings (those beginning with three single or three double quotes, as with docstrings) make for especially dramatic effects: as soon as you begin the string, the rest of the file gets recolored

† See also section 24.6 of the Python library documentation (*library/idle/idle.html* in the HTML version).

as a string—at least, until the next set of matching triple quotes is encountered. That's fine if you're still typing the string, but if the following code is still colored as a string after you think you have finished typing the string, you need to add or fix the quotes at the end of the string.

- A particularly helpful command is Show Surrounding Parens on the Edit menu, typically used by typing its keyboard shortcut. This briefly highlights the code around the cursor within the corresonding parentheses, brackets, or braces. This can help you spot mistakes and clear up confusion about the structure of expressions you are editing.
- When you press Return (Enter), IDLE will indent the new line to a column matching the current syntactic context. For example, when you hit Return after the colon at the end of the first line of a function definition, it indents the next line four spaces. If the line doesn't start in the column you expect, check the syntax of the *previous* lines. Common situations in which this occurs include:
  - When you type more right parentheses at the end of a statement than are needed to "close" the last expression of the statement. The next line will indent much further to the *left* than you expected, perhaps to column 0.
  - When you don't type enough right parentheses to close the last expression. The line will indent much further to the *right* than you expected.
- Another very useful feature of IDEs is automatic completion of partially typed Python names and filenames. Taking advantage of automatic completion can save you not just a lot of typing, but a lot of mistakes (down to either typos or misremembering Python names). Here are some autocompletion hints (be aware that it can take a while to get used to how this facility works):
  - If you pause for a few seconds after typing some characters, and there are possible completions to what you have just typed, an "autocomplete window" opens.
  - Typing a tab brings up the autocomplete window immediately.
  - If there is only one possibility, the word will be completed without the window appearing.
  - The IDLE Preferences window allows you to customize several aspects of autocompletion.

Eventually, if you end up doing heavy-duty software development, you might want to switch to a more sophisticated environment. There are a number of them to choose from, including individual efforts, open source projects, and free, "personal," and "professional" editions of commercial products. As of this writing, only a small number have been released in a version compatible with Python 3. This is another reason to use IDLE—it is part of Python itself, so the version that comes with the Python 3 installation works with Python 3.

> You can find a complete list of Python IDEs at *http://wiki.python.org/moin/IntegratedDevelopmentEnvironments*. There is no indication in that list of which IDEs support Python 3, but those last updated before 2009 definitely do not (unless the page's information is not current).

Managing Python files

- Filenames are expected to be in all lowercase and to have the extension *.py*.
- Python files may be edited in any text editor as long as they are saved in plain text format.
- Do *not* save the files you edit in Python's or IDLE's directory, for these reasons:
  - —Other people may be using the Python installation on your machine; seeing your files there may confuse them, and they may even delete or overwrite them.
  - —You may want to replace your entire Python installation; in this event, you won't want to have to figure out which files are yours and move them someplace else before you can do so.
  - —Eventually you will want to have multiple directories for your Python files—different projects, different versions of a project, etc.

  Instead, create a Python working directory wherever you feel comfortable putting it—in your *Documents* folder, for example. Then create a subdirectory for each project.
- Back up the directory containing all your Python work daily, if not more often. As you work, save your files frequently according to a set backup scheme; for instance, if working on *test.py*, save it to *test01.py*, then a bit later to *test02.py*, etc.‡ (The zero is so that when you get to version 10, file listings will still be alphabetized.) Saving multiple versions will make you less wary of experimenting, since you'll know you can always go back to an earlier version—if you decide to abandon the direction you took while working on part of your code, it is often easier to revert to an earlier version than it is to make wholesale changes to the current one.
- Files with the extension *.pyc* are intermediate files that contain "compiled" versions of the corresponding *.py* file's definitions. Python saves these after successfully importing a *.py* file for the first time, and after importing a file that has been changed since the last time it was imported. The only purpose of these files is to increase import speed—they have no effect on how fast programs actually run once their modules have all been loaded.

‡ If you happen to know how, or can get someone to show you, a better approach is simply to manage the files with a version management system such as CVS, Subversion, git, or Mercurial.

It is always safe to delete the *.pyc* files. In fact, deleting old *.pyc* files can be a good idea. You may occasionally rename or delete *.py* files and then accidentally import a module with one of the old names. The import will be successful as long as there is still a corresponding *.pyc* file—that is, your code might import a module you thought was gone. Of course, you won't have meant for your code to import a deleted or renamed module: the issue here is that it can happen in a way that doesn't produce an error message to alert you to the need to change the `import` statement.

Traps

- Assignment uses one `=`, comparison two (`==`). Using only one instead of two is a very common error. Worse, it often occurs in a context where it isn't a syntax error, in which case you are left to figure out the program's consequent misbehavior by yourself.
- If your function doesn't do anything when you call it, you probably forgot to include a `return` statement in it.
- If nothing happens when you run a file, either from the command line or within IDLE, your file contains no top-level statements that produce output. The file should end with calls to `print` or functions you've defined that print things.
- Names assigned in a function have no meaning outside that function. Two different functions can assign the same name, but because the names are in different scopes they have no relation to each other.
- Keyword arguments must appear after all the "positional" (i.e., required) arguments in a function call.
- Module names are not quoted in `import` statements, nor do their names include *.py*.
- Syntax errors are frequently caused by something wrong at the end of the line before the line shown by the error message. Common causes of these errors include:

  —Failing to end the first line of a `def` with a colon

  —Ending the previous line with too few or too many parentheses

  Before even thinking about what the error means, you should glance at the line indicated by the error message, and then the one before it. If you notice an obvious problem, fix it and try again.

Tracebacks

Here are some representative error messages:

`AssertionError: invalid argument`
: The value in an assertion statement is false; the second value in the statement, if there is one, becomes the message.

`ImportError: No module named version`
: An attempt was made to import a module named `version`, but Python couldn't find a module by that name.

`ImportError: cannot import name vision`
: An attempt was made to import a name from a module using the `from module import name` form of the `import` statement, but the module doesn't define anything with that name.

`NameError: name 'Non' is not defined`
: Python doesn't recognize a name: it's not built in and hasn't been defined with an assignment or `def` statement. Other than misspellings, two frequent causes are the use of the name in one function when it was assigned in another and the use of a name from a module without preceding it with the module's name or importing the name from the module.

`TypeError: gc_content() takes exactly 1 positional argument (0 given)`
: A call to a function supplied too few or too many positional (i.e., required, or, equivalently, nonkeyword) arguments.

CHAPTER 3
Collections

Instances of primitive Python types represent individual values, as we saw in the first chapter. Python also has built-in types that are *compound*, meaning that they group together multiple objects. These are called *collections* or *containers*. The objects in a collection are generally referred to as *items* or *elements*. Collections can hold any number of items. Some collection types can even contain items with a mixture of types, including other collections.

There are three categories of built-in collection types in Python: *sets*, *sequences*, and *mappings*. *Streams* are another kind of collection whose elements form a series in time, rather than the computer's memory space. Though they are not formally recognized in Python, many of Python's functions and methods—especially those that manipulate files—implement stream-like behavior. Modules such as `array` and `collections` provide additional collection types.

The primary distinguishing characteristic of the four collection categories is how individual elements are accessed: sets don't allow individual access; sequences use numerical *indexes*; mappings use *keys*; and streams just provide one address—"next." Strings straddle the border between primitives and collections, having characteristics of both; when treated as collections they are interpreted as sequences of one-character substrings.

Some types are *immutable*, meaning that once they are created they cannot be changed: elements may not be added, removed, or reordered. Primitive (noncollection) types are also immutable; in particular, strings cannot be changed once created. There is one very important restriction on the types of elements some collections can contain: mutable built-in collections cannot be added to sets or used as mapping keys.

You can create a collection by calling its type as a function. With no arguments, an empty collection is created. With a sequence argument, a new collection is created from the sequence's elements. Some collection types can be called with an integer argument to create an empty collection of that number of elements. Many of the collection types can also be created directly with Python syntax, the way strings are created using quotes.

Certain basic operations and functions are applicable to all collection types. These are shown in Table 3-1. The first four return a Boolean value; such functions are called *predicates*.

Table 3-1. Operations and functions that work for all collection types

Operation	Returns
`x in coll`	`True` if `coll` contains `x`.
`x not in coll`	`True` if `coll` does not contain `x`.
`any(coll)`	`True` if any item in `coll` is true, otherwise `False`.
`all(coll)`	`True` if every item in `coll` is true, otherwise `False`.
`len(coll)`	The number of items in `coll` (not supported by streams).
`max(coll[, key=function])`	The maximum item in `coll`, which may not be empty.[a]
`min(coll[, key=function])`	The minimum item in `coll`, which may not be empty.[a]
`sorted(coll[, keyfn] [, reverseflag])`	A list containing the elements of `coll`, sorted by comparing elements or, if *keyfn* is included in the call, comparing the results of calling *keyfn* for each element; if *reverseflag* is true, the ordering is reversed. *keyfn* and *reverseflg* must be specified as keyword arguments, not positionally.[a]

[a] Atypically, the optional arguments to `max`, `min`, and `sorted` may not be supplied positionally in a call—they must be provided as keyword arguments. The *keyfn* parameter is explained at the end of the chapter, in the section "Functional Parameters" on page 89.

Sets

A *set* is an *unordered* collection of items that contains *no duplicates*. The implementation enforces the restriction against duplication: attempting to add an element that a set already contains has no effect. The type *frozenset* is an immutable version of `set`. There is one important limitation on the type of objects that sets and frozensets can contain: they cannot contain instances of mutable types, or even immutable types that contain mutable elements.

Since strings behave as collections, a string can be used as the argument for a call to `set`. The resulting set will contain a single-character string for each unique character that appears in the argument. (This behavior may surprise you from time to time!) The order in which the elements of a set are printed will not necessarily bear any relation to the order in which they were added:

```
>>> set('TCAGTTAT')
{'A', 'C', 'T', 'G'}
```

Sets can also be created syntactically by enclosing comma-separated values in curly braces (`{}`). This construction is called a *set display*. There is no corresponding syntax for frozensets—you must call `frozenset` to create one. Empty braces do not create an empty set, but calling `set` with no arguments does. (Empty braces create an empty dictionary, discussed shortly.)

Python 2: The brace notation for creating sets is not supported; you must call `set` as a function to create a set.

Let's look at some examples:

```
>>> DNABases = {'T', 'C', 'A', 'G'}
>>> DNABases
{'A', 'C', 'T', 'G'}
>>> RNABases = {'U', 'C', 'A', 'G'}
>>> RNABases
{'A', 'C', 'U', 'G'}
```

When a string is inside the pair of braces that creates a set, the string is *not* broken up into its individual characters:

```
>>> {'TCAG'}
{'TCAG'}
>>> {'TCAG', 'UCAG'}
{'UCAG', 'TCAG'}
>>>
```

Sets can be compared as shown in Table 3-2. The operator versions require both operands to be sets, but the method versions allow the argument to be any collection.

Table 3-2. Set comparison operations

Operator	Method	Returns
	set1.isdisjoint(*coll*)	True if the set and the argument have no elements in common
set1 <= *set2*	*set1*.issubset(*coll*)	True if every element of *set1* is also in *set2* (*coll* for the method)
set1 < *set2*		True if every element of *set1* is also in *set2* (*coll* for the method) and *set2* is larger than *set1* (i.e., *set1* is a proper subset of *set2*)
set1 >= *set2*	*set1*.issuperset(*coll*)	True if every element of *set2* (*coll* for the method) is also in *set1*
set1 > *set2*		True if every element of *set2* is also in *set1* and *set1* is larger than *set2* (i.e., *set2* is a proper subset of *set1*)

Python's `set` and `frozenset` types implement the mathematical operations of set algebra. These use regular operator symbols and, in some cases, equivalent method calls. As with the comparison operations, the operator versions require both operands to be sets, while the methods allow the argument to be any collection. Table 3-3 shows the operators and the corresponding methods.

Table 3-3. Algebraic set operations

Operator	Method	Returns
set1 | *set2*	*set1*.union(*coll*, ...)	A new set with the elements of both *set1* and *set2* (or *set1* and each of the arguments)
set1 & *set2*	*set1*.intersection(*coll*, ...)	A new set with the elements that are common to *set1* and *set2* (or *set1* and each of the arguments)
set1 - *set2*	*set1*.difference(*coll*, ...)	A new set with the elements in *set1* that are not in *set2* (or any of the arguments)
set1 ^ *set2*	*set1*.symmetric_difference(*coll*)	A new set with the elements that are in either *set1* or *set2* (or each of the arguments) but not both

Using sets, we can rewrite `validate_base_sequence` more succinctly than with the ponderous conditional expressions used in the last chapter. Example 3-1 illustrates.

Example 3-1. Rewriting validate_base_sequence using a set

```
DNAbases = set('TCAGtcag')
RNAbases = set('UCAGucag')
def validate_base_sequence(base_sequence, RNAflag = False):
    """Return True if the string base_sequence contains only upper- or lowercase
    T (or U, if RNAflag), C, A, and G characters, otherwise False"""
    return set(base_sequence) <= (RNAbases if RNAflag else DNAbases)
```

We could have just written out the set values directly, enclosing eight one-character strings within braces, either in the assignment statements or instead of the assigned names in the function definition. Using `set` to convert a string meant less typing, though, which is always good. A fundamental fact about programming—or any computer use, for that matter—is that every character you type is a chance to make a mistake!

Not only is this version of the function definition more succinct, but it more directly expresses the function's purpose. The function tests whether all the characters in its argument are in the set of allowable base characters. This definition says that explicitly, whereas the previous definition counted the number of times each base occurred and compared the sum of those counts to the length of the sequence. It worked, but it was a roundabout way of saying something that can be said more simply and clearly.

The algebraic set operations produce new sets. There is also a group of operators and methods for modifying sets, shown in Table 3-4. The operator forms require both operands to be sets, while the methods can take any kind of collection argument.

The assignment statements in Example 3-1 were placed before the function definition, so they only get executed once rather than every time the function is called. It's not likely to matter much in this case, but keep in mind that in general you should avoid repeating computations unnecessarily where convenient. You should never let "efficiency" considerations get in your way while developing code, since you won't know how fast your program runs or which parts are slow—or even whether you'll end up using the part of the code whose efficiency concerns you—until the code is working. That said, it's easy enough to keep simple, obvious things like assignment statements whose values don't change out of loops as a matter of practice.

Table 3-4. Set update operations

<table>
<tr><th>Operator</th><th>Method</th><th>Result</th></tr>
<tr><td>set1 |= set2</td><td>set1.update(coll)</td><td>Updates set1 by adding the elements in set2 (coll for the method)</td></tr>
<tr><td>set1 &= set2</td><td>set1.intersection_update(coll)</td><td>Updates set1 to keep only the elements that are in both set1 and set2 (coll for the method)</td></tr>
<tr><td>set1 -= set2</td><td>set1.difference_update(coll)</td><td>Updates set1 to keep only the elements that are in set1 but not in set2 (coll for the method)</td></tr>
<tr><td>set1 ^= set2</td><td>set1.symmetric_difference_update(coll)</td><td>Updates set1 to keep only the elements that are in either set1 or set2 (coll for the method)</td></tr>
<tr><td></td><td>set1.add(item)</td><td>Adds item to set1</td></tr>
<tr><td></td><td>set1.remove(item)</td><td>Removes item from set1; it is an error if item is not in the set</td></tr>
<tr><td></td><td>set1.discard(item)</td><td>Removes item from set1 if it is present; no error if it is not present</td></tr>
</table>

Sequences

Sequences are *ordered* collections that *may contain duplicate elements*. Unfortunately, the word "sequence" is frequently used in both Python programming and bioinformatics, not to mention in ordinary writing. Hopefully, the context will make its intended meaning clear.

Because sequences are ordered, their elements can be referenced by position. We've already seen how this works with string indexing and slicing; the same mechanisms work with other sequence types. Table 3-5 summarizes the various cases of slicing expressions. Remember that a negative index counts from just past the last element.

Table 3-5. Summary of sequence slicing

Expression	Returns
`seq[i:j]`	Elements of *seq* from *i* up to, but not including, *j*
`seq[i:]`	Elements of *seq* from *i* through the end of the sequence
`seq[:j]`	Elements of *seq* from the first up to, but not including, *j*
`seq[:-1]`	Elements of *seq* from the first up to, but not including, the last
`seq[:]`	All the elements of *seq*—i.e., a copy of *seq*
`seq[i:j:k]`	Every k^{th} element of *seq*, from *i* up to, but not including, *j*

There are six built-in sequence types, as summarized in Table 3-6.

Table 3-6. Sequence types

Type	Element type	Mutable?	Syntax
`str`	One-character strings[a]	No	Quotes of various kinds
`bytes`	8-bit bytes[b]	No	A string with a b prefix
`bytearray`	8-bit bytes	Yes	None
`range`	Integers	No	None
`tuple`	Any	No	Parentheses
`list`	Any	Yes	Brackets

[a] Strings don't really "contain" characters or one-character strings, but for many purposes they behave as if they did.

[b] Bytes are equivalent to the integers from 0 through 255.

Python 2: Substantial changes were made to the sequence types from Python 2.x to 3.x. The most significant change was that Python 2.x strings contain one-byte "ASCII" characters but Python 3.x strings contain "Unicode" characters. In Python 2 `range` was a function, but in Python 3 it is a type, replacing what in Python 2 was the type `xrange`. Several other sequence types were also replaced by new ones with different characteristics.

Table 3-7 lists the operations that are supported by all sequence types, with some clarifying details.

Table 3-7. Generally supported by sequence operations

Operation	Restriction
`<`, `<=`, `>`, `>=`	Same type operands, except that `bytes` and `bytearray` may be compared
`==`, `!=`	Any type operands, but always `False` if the operand types are different
`in`, `not in`	
Repetition (`*`)	

Operation	Restriction
Concatenation (+)	Same type operands, except that a `bytes` and a `bytearray` may be concatenated, with the type of the result the type of the lefthand operand
Indexing	
Slicing	

All sequence types except for ranges and files support the methods `count` and `index`. In addition, all sequence types support the `reversed` function, which returns a special object that produces the elements of the sequence in reverse order.

Strings, Bytes, and Bytearrays

Strings are sequences of Unicode characters (though there is no "character" type). Unicode is an international standard* that assigns a unique number to every character of all major languages (including special "languages" such as musical notation and mathematics), as well as special characters such as diacritics and technical symbols. Unicode currently defines more than 100,000 characters. The 128 "ASCII" characters (digits, lower- and uppercase letters, punctuation, etc.) have the same numerical values in both plain ASCII and Unicode representations.

The `bytes` and `bytearray` types are sequences of single bytes. They have essentially the same operations and methods as `str`. One important difference is that with a string, regardless of whether you index an element or specify a slice, you always get back a string. With `bytes` and `bytearray`s, although slices return the same type, indexing an element returns an integer from `0` through `255`.

Another interesting difference is that when either `bytes` or `bytearray` is called with an integer argument, it creates an instance of that length containing all `0`s. If called with a sequence of integers with values from `0` through `255`, it performs the normal "conversion" kind of instance creation, creating a `bytes` or `bytearray` object with those integers as its elements. Note that strings do *not* contain integers—their components are single character strings—so they can't be the argument to `bytes` or `bytearray`.

There are some differences in how each sequence type treats the `in` and `not in` operations. When the righthand operand is of type `str`, the lefthand operator must also be a `str`, and the operation is interpreted as a substring test. When the righthand operand is of type `bytes` or `bytearray`, the lefthand side can be an integer from `0` to `255`, or it can be a `bytes` or a `bytearray`. If it is an integer, `in` and `not in` are membership tests; otherwise, they are subsequence tests, as with `str`.

The `bytes` and `bytearray` types serve many purposes, one of which is to efficiently store and manipulate raw data. For instance, a piece of lab equipment might assign a signal

* See *http://www.unicode.org/standard/WhatIsUnicode.html* and *http://www.unicode.org/standard/principles.html* for more information.

strength value of between `0` and `255` for something it is measuring repetitively. The natural way to store these values in Python is to use `bytes`. Data obtained from outside a program in other ways is often in the form of raw bytes. (We'll see this when we discuss database access, web downloads, and other such topics later in the book.) That is, the bytes meant something where the data came from, but they have no intrinsic meaning inside Python. Interpretation of externally obtained bytes is the program's responsibility.

Strings, bytes, and bytearrays have a very wide range of application in the management and manipulation of data and human-readable text. They provide a much richer repertoire of methods than other sequence types. The more commonly used ones are listed in the following sections. The sections refer only to strings, but bytes and bytearrays support the same methods. Other methods will be discussed a little later, in the section "Sequence-oriented string methods" on page 65.

Creating

We know that a new empty collection can be created by using the name of its type in a call with no arguments. Some collection types can also be called with arguments. Following are the creation calls for `str`, `bytes`, and `bytearray` objects:

`str()`
: Returns an empty string

`str(`*`obj`*`)`
: Returns a printable representation of *`obj`*, as specified by the definition of the type of *`obj`*

`bytes()`
: Returns an empty `bytes` object

`bytes(`*`n`*`)`
: With *`n`* an integer, returns a `bytes` object with *`n`* zeros

`bytes(`*`integer-sequence`*`)`
: With *`integer-sequence`* a sequence containing integers between `0` and `255` inclusive, returns a `bytes` object with the corresponding integers

`char(`*`n`*`)`
: Returns the one-character string corresponding to the integer *`n`* in the Unicode system

`ord(`*`char`*`)`
: Returns the Unicode number corresponding to the one-character string *`char`*

The functions for creating `bytearray` objects work the same way as the corresponding functions for `bytes`.

Testing

It is often necessary in processing a string to determine what kinds of characters it "contains." These methods provide convenient ways to determine that for the most common cases:

`str1.isalpha()`
: Returns true if *str1* is not empty and all of its characters are alphabetic

`str1.isalnum()`
: Returns true if *str1* is not empty and all of its characters are alphanumeric

`str1.isdigit()`
: Returns true if *str1* is not empty and all of its characters are digits

`str1.numeric()`
: Returns true if *str1* is not empty and all of its characters are numeric, including Unicode number values and names

`str1.isdecimal()`
: Returns true if *str1* is not empty and all of its characters are characters that can be used in forming decimal-radix numbers, including Unicode number values and names

`str1.islower()`
: Returns true if *str1* contains at least one "cased" character and all of its cased characters are lowercase

`str1.isupper()`
: Returns true if *str1* contains at least one "cased" character and all of its cased characters are uppercase

`str1.istitle()`
: Returns true if *str1* is not empty, all of its uppercase characters follow "uncased" characters (spaces, punctuation, etc.), and all of its lowercase characters follow cased characters

Searching

The following methods search for one string inside another. In the descriptions that follow, *startpos* and *endpos* function as they do in slices:

`str1.startswith(str2[, startpos, [endpos]])`
: Returns true if *str1* starts with *str2*

`str1.endswith(str2[, startpos, [endos]])`
: Returns true if *str1* ends with *str2*

`str1.find(str2[, startpos[, endpos]])`
: Returns the lowest index of *str1* at which *str2* is found, or `-1` if it is not found

str1.rfind(*str2*[, *startpos*[, *endpos*]])
: Performs a reverse find: returns the *highest* index where *str2* is found in *str1*, or -1 if it is not found

str1.index(*str2*[, *startpos*[, *endpos*]])
: Returns the lowest index of *str1* at which *str2* is found, or ValueError if it is not found

str1.rindex(*str2*[, *startpos*[, *endpos*]])
: Returns the *highest* index of str1 at which str2 is found, or ValueError if it is not found

str1.count(*str2*[, *startpos*[, *endpos*]])
: Returns the number of occurrences of *str2* in *str1*

Replacing

Methods that return a new string with parts of the old string replaced with something else form the basis of a lot of code. The replace method is used particularly frequently, but there are also two other methods that constitute a powerful little facility (though they aren't used all that much):

str1.replace(*oldstr*, *newstr*[, *count*])
: Returns a copy of *str1* with all occurrences of the substring *oldstr* replaced by the string *newstr*; if *count* is specified, only the first *count* occurrences are replaced.

str1.translate(*dictionary*)
: With *dictionary* having integers as keys, returns a copy of *str1* with any character *char* for which ord(*char*) is a key in *dictionary* replaced by the corresponding value. Exactly what the replacement does depends on the type of the value in the dictionary, as follows:

  None
  : Character is removed from *str1*

  Integer n
  : Character is replaced by chr(*n*)

  String str2
  : Character is replaced by *str2*, which may be of any length

str.maketrans(*x*[, *y*[, *z*]])
: (Called directly through the str type, not an individual string.) Produces a translation table for use with translate more conveniently than manually constructing the table. Arguments are interpreted differently depending on how many there are:

  x
  : *x* is a dictionary like that expected by translate, except that its keys may be either integers or one-character strings.

x, *y*
: *x* and *y* are strings of equal length; the table will translate each character of *x* to the character in the corresponding position of *y*.

x, *y*, *z*
: As with two arguments, plus all characters in the string *z* will be translated to *x* (i.e., removed).

Changing case

The methods listed in this section return a new string that is a copy of the original with its characters converted as necessary to a specified case:

`str1.lower()`
: Returns a copy of the string with all of its characters converted to lowercase

`str1.upper()`
: Returns a copy of the string with all of its characters converted to uppercase

`str1.capitalize()`
: Returns a copy of the string with only its first character capitalized; has no effect if the first character is not a letter (e.g., if it is a space)

`str1.title()`
: Returns a copy of the string with each word beginning with an uppercase character and the rest lowercase

`str1.swapcase()`
: Returns a copy of the string with lowercase characters made uppercase and vice versa

Reformatting

Each of the methods in this group returns a string that is a copy of the original after applying text formatting operations:

`str1.lstrip([chars])`
: Returns a copy of *str1* with leading characters removed. *chars* is a string that includes the characters to be removed (any combination of them); if it is omitted or `None`, the default is whitespace.

`str1.rstrip([chars])`
: Returns a copy of *str1* with trailing characters removed. *chars* is a string that includes the characters to be removed (any combination of them); if it is omitted or `None`, the default is whitespace.

`str1.strip([chars])`
: Returns a copy of *str1* with leading and trailing characters removed. *chars* is a string that includes the characters to be removed (any combination of them); if it is omitted or `None`, the default is whitespace.

str1.ljust(*width*[, *fillchar*])
: Returns ***str1*** left-justified in a new string of length ***width***, "padded" with ***fillchar*** (the default fill character is a space).

str1.rjust(*width*[, *fillchar*])
: Returns ***str1*** right-justified in a new string of length ***width***, "padded" with ***fillchar*** (the default fill character is a space).

str1.center(*width*[, *fillchar*])
: Returns ***str1*** centered in a new string of length ***width***, "padded" with ***fillchar*** (the default fill character is a space).

str1.expandtabs([*tabsize*])
: Returns ***str1*** with all of its tabs replaced by one or more spaces according to ***tabsize*** (the default is `8`) and the column at which the tab occurs on its line.

The format method and function

Many Python programs produce formatted text output. While the methods described in the preceding sections provide a substantial set of operations for manipulating individual strings, arranging the output of several pieces of information together can be a rather intricate project. Python includes a very powerful general formatting facility that can be invoked either through the built-in function called `format` or through a string method of the same name. These can save you a lot of effort:

format(*value*[, *format-specification*])
: Returns a string obtained by formatting ***value*** according to the ***format-specification***; if no ***format-specification*** is provided, this is equivalent to `str(`*value*`)`

format-specification.format(*posargs*, ..., *kwdargs*, ...)
: Returns a string formatted according to the ***format-specification***; any number of positional arguments may be followed by any number of keyword arguments

Format specifications incorporate values where curly braces occur. A pair of braces can contain a number (to indicate a positional argument), a name (to indicate a keyword argument), or neither (to indicate the next positional argument). However, empty braces cannot appear in a format string after braces containing numbers or names. The advantage of using names in format specifications is the same as that of using keyword arguments in function definitions: they describe the role the value plays in the formatted string, and they allow the arguments to be specified in any order.

Given:

```
str1 = 'a string'
```

the following all produce the result `'"a string" contains 8 characters'`:

```
'"{0}" contains {1} characters'.format(str1, len(str1))
'"{}" contains {} characters'.format(str1, len(str1))
'"{string}" contains {length} characters'.format(string=str1,
length=len(str1))
'"{string}" contains {length} characters'.format(length=len(str1),
string=str1)
```

The `format` method and function can do more than simply replace values. The format specification dictates how each value is formatted within the resulting string. A colon inside a pair of braces begins a format specifier. The complete general structure of a format specifier is as follows:

```
[[fill]align][sign][#][0][width][.precision][type]
```

Some of these you will use only rarely, but when you need them they give you all the control you need over the format of a string. The values for the `fill`, `align`, and `width` fields are listed in Table 3-8. The numeric parts of the specifier are described in Table 3-9, and Table 3-10 gives the options for integer and floating-point type specifications.

Table 3-8. General format specifier fields

Field	Option	Meaning
`align`	`<`	Align left (the default)
	`>`	Align right
	`^`	Center
	`=`	Padding goes *between* sign and digits (for numeric values only)
`fill`	`not }`	Fill character used for aligning; if the second character of the specifier is not a fill option, the first character is not interpreted as a fill character
`width`	`integer`	Minimum field width

Table 3-9. Numeric format specifier fields

Field	Option	Meaning
`sign`	`+`	Indicates that signs should be used for both positive and negative numbers
	`-`	Indicates that a sign should be used only for negative numbers (the default)
	`space`	Indicates that a leading space should be used for positive numbers
`#`		Only for binary, octal, or hexadecimal output of integers: prefix the number with `0b`, `0o`, or `0x`, respectively
`0`	`zero-padding`	Preceding `width`, indicates zero-padding; equivalent to alignment `=` and fill character `0`
`precision`	`integer`	The number of digits to be displayed after the decimal point of a number formatted with `f` or `F` or before and after the decimal point of a number formatted with `g` or `G`

Table 3-10. Numeric type specifier fields

Field	Option	Meaning
Types for integers	b	Binary (base 2)
	c	Character: converts the integer to the corresponding Unicode character
	d	Decimal digit (base 10; the default)
	o	Octal (base 8)
	x	Hexadecimal (base 16), using lowercase letters a–f
	X	Hexadecimal (base 16), using uppercase letters A–F
	n	Like d but uses the locally appropriate number separators
Types for floating-point and decimal values	e	Exponent notation (scientific notation using e for the exponent)
	E	Same as e but uses uppercase E for the exponent
	f	Fixed point
	F	Same as f
	g	"General" format (the default): prints the number as fixed point unless it is too large, in which case it switches to e
	G	Same as g but uses E
	n	Like g but uses the locally appropriate number separators

Ranges

A *range* represents a series of integers. We'll see some uses for ranges later in this chapter and also in the next. They may not look like much, but their use is quite fundamental to Python programming.

With one argument, `range(`*`stop`*`)` creates a `range` representing the integers from `0` up to but not including *`stop`*. With two arguments, `range(`*`start, stop`*`)` creates a `range` representing the integers from *`start`* up to but not including *`stop`*. (If *`stop`* is less than or equal to *`start`*, the range will be empty.) With three arguments, `range(`*`start, stop, step`*`)` creates a `range` representing the integers from *`start`* up to but not including *`stop`*, in increments of *`step`*. Normally, if *`step`* is negative, *`stop`* will be less than *`start`*. Note the similarity between ranges and slice expressions. Here are some usage examples:

```
>>> range(5)
range(0, 5)
>>> set(range(5))
{0, 1, 2, 3, 4}
>>> range(5, 10)
range(5, 10)
>>> set(range(5, 10))
{8, 9, 5, 6, 7}
>>> range(5, 10, 2)
range(5, 10, 2)
>>> set(range(5, 10, 2))
```

```
{9, 5, 7}
>>> set(range(15, 10, -2))
{15, 13, 11}
>>> set(range(0, -25, -5))
{0, -15, -5, -20, -10}
```

Tuples

A *tuple* is an *immutable* sequence that can contain *any type of element*. A very common use of tuples is as a simple representation of pairs such as x, y coordinates. For example, the built-in function `divmod(x,y)` returns a two-element tuple containing the quotient and the remainder of the integer division of *x* by *y*. Tuples are also the natural form for record-oriented data obtained from an external source, such as a row in a database.

Tuple syntax

Tuples are written as comma-separated series of items surrounded by parentheses. This leads to the only ambiguity in Python's syntax: since parentheses are used to group expressions, a parenthesized expression such as `(3 * 5)` cannot be read as a one-element tuple. To get around this problem, one-element tuples must be written with a comma after their single element. (Commas may always follow the last element of any sequence display.) An empty pair of parentheses creates an empty tuple. Here are some examples:

```
>>> ('TCAG', 'UCAG')          # a two-element tuple
('TCAG', 'UCAG')
>>> ('TCAG',)                 # a one-element tuple
('TCAG',)
>>> ()                        # an empty tuple
()
>>> ('TCAG')                  # not a tuple!
'TCAG'
```

Given a sequence as an argument, the `tuple` function creates a tuple containing the elements of the sequence. Remember, a string is considered a sequence of one-character strings in contexts like this:

```
>>> tuple('TCAG')
('T', 'C', 'A', 'G')
>>> tuple(range(5,10))
(5, 6, 7, 8, 9)
```

Tuple packing and unpacking

The righthand side of an assignment statement can be a series of comma-separated expressions. The result is a tuple with those values. This is called *tuple packing*:

```
>>> bases = 'TCAG', 'UCAG'    # a two-element tuple
>>> bases
('TCAG', 'UCAG')
```

Multiple expressions in a `return` statement are also packed into a tuple, so this is a very convenient way for a function to return more than one value. Example 3-2 illustrates

a multiple return from a function. It uses the `recognition_site` function defined in the preceding chapter to simulate the enzyme's cut.

Example 3-2. Returning multiple values from a function

```
def restriction_cut(base_seq, recognition_seq, offset = 0):
    """Return a pair of sequences derived from base_seq by splitting it at the first appearance
    of recognition_seq; offset, which may be negative, is the number of bases relative to the
    beginning of the site where the sequence is cut"""
    site = recognition_site(base_seq, recognition_seq)
    return base_seq[:site+offset], base_seq[site+offset:]
```

A tuple—either explicitly enclosed in parentheses or packed—can be the lefthand side of a simple assignment statement (but not an augmented assignment statement, such as one that uses `+=`). The righthand side of the assignment statement must be a sequence containing the same number of elements as the tuple on the left, either as an implicitly packed tuple or a single sequence value. For example:

```
>>> DNABases, RNABases = 'TCAG', 'UCAG'
>>> DNABases
'TCAG'
>>> RNABases
'UCAG'
>>>
```

A tuple on the left and a tuple-returning function on the right is a fairly common construction. Sometimes a single name will be assigned to the tuple result, but often a series of names will be assigned to the unpacked values that are returned:

```
>>> aseq1 = 'AAAAATCCCGAGGCGGCTATATAGGGCTCCGGAGGCGTAATATAAAA'
>>> left, right  = restriction_cut(aseq1, 'TCCGGA')
>>> left
'AAAAATCCCGAGGCGGCTATATAGGGC'
>>> right
'TCCGGAGGCGTAATATAAAA'
>>>
```

An interesting consequence of this is that the bindings of two names can be exchanged in one statement. This kind of situation does arise from time to time. This works because all the values on the right are evaluated, then packed into a tuple, and finally unpacked for assignment on the left:

```
>>> left, right = right, left
```

Lists

A *list* is a *mutable sequence* of *any kind of element*. Lists are a highly flexible and widely applicable kind of container—so much so that they could be considered the archetypal collection type. One way of thinking of lists is as mutable tuples, since anything you can do with a tuple you can also do with a list. However, you can do a lot more with lists than you can with tuples.

The syntax for lists is a comma-separated series of values enclosed in square brackets. (Square brackets do not lead to the kind of ambiguity with a one-element list that parentheses do with tuples.)

A few built-in functions return lists. For example, `dir(x)` returns a list of the names associated with whatever *x* is. As with `help(x)`, the list will contain more than you probably want to see: it includes all the names beginning with underscores, which are meant to be internal to *x*'s implementation. Later in this chapter (Example 3-4), we'll define a function called `dr` that filters the result of calling `dir` to omit the private names.

There are many operations and methods that modify lists. Even slicing can be used to change what a list contains. Lists are often used as a way of producing a modified copy of an immutable sequence. First, a new list is created by calling `list` with an immutable sequence as its argument; then the list is modified, and finally, the type of the immutable sequence is called with the modified list as its argument. As you'll see in the next chapter, another common use of lists is to accumulate values.

Bytearrays are essentially lists with elements restricted to the integers `0` to `255` inclusive. All the operations and methods that you can use for modifying a list can also be used to modify a bytearray.

Statements that modify lists

An index expression designates a particular element of a sequence. A slice specifies a subsequence. Index and slice expressions of lists and bytearrays—the two mutable sequence types—can appear on the lefthand side of an assignment statement. What this means is that the element or subsequence represented by the index or slice expression is replaced by the value of the expression on the assignment statement's righthand side. The change is made to the list itself, not to a copy (the way it would be with string operations).

Slice assignments can express many kinds of list manipulations, as shown in Table 3-11. Some can even change the size of the list. It is well worth studying their varieties.

Table 3-11. List modification assignments

Assignment expression	Result
`lst[n] = x`	Replaces the nth element of `lst` with *x*
`lst[i:j] = coll`	Replaces the ith through j^{th} elements of `lst` with the elements of `coll`
`lst[i:j] = any_empty_collection`	Deletes the ith through j^{th} elements of `lst` (an important special case of `lst[i:j] = seq`)
`lst[i:j:k] = coll`	Replaces the elements of `lst` designated by the slice with the elements of `coll`, whose length must equal the number of elements designated by the slice
`lst[n:n] = coll`	Inserts the elements of `coll` before the nth element of `lst`
`lst[len(lst):len(lst)] = [x]`	Adds *x* to the end of `lst`

Assignment expression	Result
lst[len(*lst*):len(*lst*)] = *coll*	Adds the elements of *coll* at the end of *lst*
lst += *coll*	
lst[:] = *coll*	Replaces the entire contents of *lst* with the elements of *coll*

There is a simple statement that can remove elements: `del`, for "delete."

STATEMENT

Deletion

The `del` statement removes one or more elements from a list or bytearray.

```
del lst[n]        # remove the nth element from lst
del lst[i:j]      # remove the ith through jth elements from lst
del lst[i:j:k]    # remove every k elements from i up to j from lst
```

List modification methods

Table 3-12 shows the methods that change a list. These methods are unusual in that they actually change the list itself, rather than producing a modified copy as would similar methods of other types. They are also unusual because—with the exception of `pop`—they do not return a value (i.e., they return `None`).

Table 3-12. List modification methods

Method	Result
lst.append(*x*)	Adds *x* to the end of *lst*
lst.extend(*x*)	Adds the elements of *x* at the end of *lst*
lst.insert(*i*, *x*)	Inserts *x* before the *i*th element of *lst*
lst.remove(*x*)	Removes the first occurrence of *x* from *lst*; an error is raised if *x* is not in *lst*
lst.pop([*i*])	Removes the *i*th element from *lst* and returns it; if *i* is not specified, removes the last element
lst.reverse()	Reverses the list
lst.sort([*reverseflag*] [, *keyfn*])	Sorts the list by comparing elements or, if *keyfn* is included in the call, comparing the results of calling *keyfn* for each element; if *reverseflag* is true, the ordering is reversed; *keyfn* and *reverseflg* must be specified as keyword arguments, not positionally[a]

[a] Atypically, the optional arguments to `sort` may not be supplied positionally. Either or both may be supplied, but only as keyword arguments. The parameter `reverse` is simply a flag that controls whether the list is sorted in increasing or decreasing order. The *keyfn* parameter is explained at the end of the chapter, in the section "Functional Parameters" on page 89.

Sets, frozensets, strings, tuples, and lists can be concatenated with other values of the same type, and the result is a new value of the same type. List modifications are different. No new list is created; instead, the contents of the original list are changed. This is particularly evident in the different results obtained by concatenating two lists as

opposed to using the `extend` method, as demonstrated in the following interaction and Figure 3-1:

```
>>> list1 = [1,2,3]
>>> list2 = [4,5]
>>> list1 + list2              # concatenation
[1, 2, 3, 4, 5]                # produces a new list
>>> list1
[1, 2, 3]                      # while list1 remains unchanged
>>> list2
[4, 5]                         # as does list2
>>> list1.extend(list2)        # extension
>>> list1
[1, 2, 3, 4, 5]                # modifies list1
>>> list2
[4, 5]                         # but not list2
```

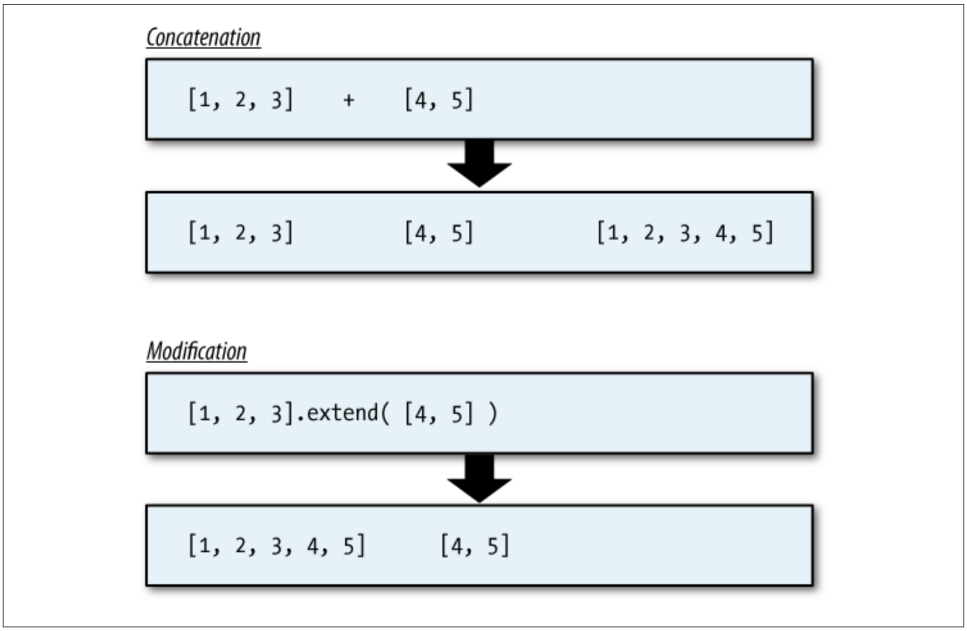


Figure 3-1. List concatenation versus list modification

Augmented assignment of one list to another *modifies* the list on the lefthand side of the statement.

Sequence-oriented string methods

There are some important string methods that we haven't looked at yet; they were not included earlier in the chapter because we hadn't yet discussed the types of sequences they take as arguments or return as results. In practice the sequence arguments are

usually lists, but in principle they can be any kind of sequence. Remember too that when a sequence argument is expected, a string is interpreted as a sequence of one-character strings. Here are the remaining string methods:

`string.splitlines([keepflg])`
: Returns a list of the "lines" in ***string***, splitting at end-of-line characters. If ***keepflg*** is omitted or is false, the end-of-line characters are not included in the lines; otherwise, they are.

`string.split([sepr[, maxwords]])`
: Returns a list of the "words" in ***string***, using ***sepr*** as a word delineator. In the special case where ***sepr*** is omitted or is `None`, words are delineated by any consecutive whitespace characters; if ***maxwords*** is specified the result will have at most ***maxwords*** `+1` elements.

`string.rsplit([sepr[, maxwords]])`
: Performs a reverse split: same as `split` except that if ***maxwords*** is specified and its value is less than the number of words in ***string*** the result returned is a list containing the *last* `maxwords+1` words.

`sepr.join(seq)`
: Returns a string formed by concatenating the strings in ***seq*** separated by ***sepr***, which can be any string (including the empty string).

`string.partition(sepr)`
: Returns a tuple with three elements: the portion of ***string*** up to the first occurrence of ***sepr***, ***sepr***, and the portion of ***string*** after the first occurrence of ***sepr***. If ***sepr*** is not found in ***string***, the tuple is `(string, '', '')`.

`string.rpartition(sepr)`
: Returns a tuple with three elements: the portion of ***string*** up to the last occurrence of ***sepr***, ***sepr***, and the portion of ***string*** after the last occurrence of ***sepr***. If ***sepr*** is not found in ***string***, the tuple is `('', '', string)`.

Mappings

A *mapping* is a *mutable unordered* collection of *key/value pairs*.† Computer scientists use a number of other names to refer to data structures implementing mappings, including *associative arrays*, *lookup tables*, and *hash tables*. A physical dictionary is a real-world example of a mapping: given a word, you get a definition. Figure 3-2 illustrates the concept.

† The term "mapping" comes from mathematics, where it represents a function from a "domain" of values to a "range" of values.

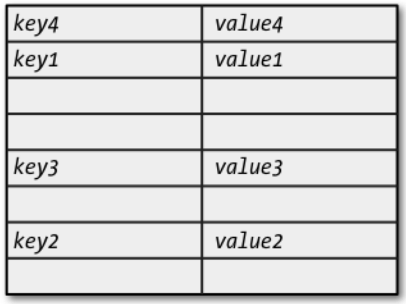


Figure 3-2. A dictionary

Dictionaries

Dictionaries have many applications in programming. There are reasons for Python to have sets, frozensets, bytes, bytearrays, and ranges, but you could easily write most of your programs using just strings, lists, and dictionaries. There is just one mapping type in Python: `dict`, for "dictionary." Like other collection types, `dict` can be called with a collection argument to create a dictionary with the elements of the argument. However, those elements must be tuples or lists of two elements—a key and a value:

```
dict((('A', 'adenine'),
      ('C', 'cytosine'),
      ('G', 'guanine'),
      ('T', 'thymine')
     ))
```

Because they are so frequently used, Python provides a notation for dictionaries similar to that of sets:‡ a comma-separated list of key/value pairs enclosed in curly braces, with each key and value separated by a colon. You can put spaces before, after, or both before and after the colon. They are printed with no space before and a space after the colon, so you might want to choose that style. Empty braces create an empty dictionary (not an empty set). The order within the braces doesn't matter, since the dictionary implementation imposes its own order. A simple example demonstrates that:

```
>>> {'A': 'adenine', 'C': 'cytosine', 'G': 'guanine', 'T': 'thymine'}
{'A': 'adenine', 'C': 'cytosine', 'T': 'thymine', 'G': 'guanine'}
```

The keys of a mapping must be unique within the collection, because the dictionary has no way to distinguish different values indexed by the same key. As with sets, the implementation of `dict` does not allow keys to be instances of mutable built-in types.

‡ You could consider a set to be a dictionary where the value associated with each key is the key itself; in fact, they are often implemented that way in various languages and libraries.

Even a tuple used as a key may not contain mutable built-in types. Any other kind of object may be used as a key, including numbers, strings, and frozensets.

What happens if the argument added to a set or a key used with a dictionary is a collection with mutable elements? Here's an example:

```
>>> set([['T', 'C', 'A', 'G'],          # attempt to create a set
         ['U', 'C', 'A', 'G']])         # from a list of two lists
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

"Hashable" refers to the computation performed as part of storing items in lists and dictionaries so that they can be retrieved extremely efficiently. That's all you need to know about it.

Error Message Vocabulary

Error messages developed for a programming language are phrased using a reasonable technical vocabulary, but not one beginning users—even if they are experienced programmers—will necessarily understand. (Even if you're familiar with the concept that a term refers to, you may not recognize it because you may know it by a different name.) Some error messages are pretty clear; for example, `NameError: name 'Non' is not defined`. Others are more "opaque"—you'll learn what they mean, but you may not understand why they say what they do.

As you gain experience, you'll develop your own mental dictionary that uses these error messages as keys and has explanations as values. In this case, you just have to learn that "unhashable" refers to the technical limitation that prevents mutable built-in collections from being used as elements of sets or keys of dictionaries. Summaries of some of these translations will be provided at the end of the chapters in which they appear.

Dictionary example: RNA codon translation table

Dictionaries are the natural Python representation for tabular data. For example, they can be used to hold rows obtained from a database table: the table's primary key is the dictionary's key, and the value can be a tuple, with each element of the tuple representing one column of the table.[§] (There is nothing strange about a dictionary key being part of the value associated with it—in fact, this is quite common.)

Table 3-13 shows the usual RNA codon-to-amino acid translation table. Example 3-3 shows how to represent this as a Python dictionary.

§ If you are not familiar with these database terms, you might want to read the section "Database Tables" on page 360.

Table 3-13. The RNA amino acid translation table

		Second Base			
First Base	**U**	**C**	**A**	**G**	**Third Base**
U	Phe	Ser	Tyr	Cys	**U**
	Phe	Ser	Tyr	Cys	**C**
	Leu	Ser	*stop*	*stop*	**A**
	Leu	Ser	*stop*	Trp	**G**
C	Leu	Pro	His	Arg	**U**
	Leu	Pro	His	Arg	**C**
	Leu	Pro	Gln	Arg	**A**
	Leu	Pro	Gln	Arg	**G**
A	Ile	Thr	Asn	Ser	**U**
	Ile	Thr	Asn	Ser	**C**
	Ile	Thr	Lys	Arg	**A**
	Met/*start*	Thr	Lys	Arg	**G**
G	Val	Ala	Asp	Gly	**U**
	Val	Ala	Asp	Gly	**C**
	Val	Ala	Glu	Gly	**A**
	Val	Ala	Glu	Gly	**G**

Example 3-3. The RNA translation table as a Python dictionary

```
RNA_codon_table = {
#                        Second Base
#        U             C             A             G
# U
    'UUU': 'Phe', 'UCU': 'Ser', 'UAU': 'Tyr', 'UGU': 'Cys',     # UxU
    'UUC': 'Phe', 'UCC': 'Ser', 'UAC': 'Tyr', 'UGC': 'Cys',     # UxC
    'UUA': 'Leu', 'UCA': 'Ser', 'UAA': '---', 'UGA': '---',     # UxA
    'UUG': 'Leu', 'UCG': 'Ser', 'UAG': '---', 'UGG': 'Urp',     # UxG
# C
    'CUU': 'Leu', 'CCU': 'Pro', 'CAU': 'His', 'CGU': 'Arg',     # CxU
    'CUC': 'Leu', 'CCC': 'Pro', 'CAC': 'His', 'CGC': 'Arg',     # CxC
    'CUA': 'Leu', 'CCA': 'Pro', 'CAA': 'Gln', 'CGA': 'Arg',     # CxA
    'CUG': 'Leu', 'CCG': 'Pro', 'CAG': 'Gln', 'CGG': 'Arg',     # CxG
# A
    'AUU': 'Ile', 'ACU': 'Thr', 'AAU': 'Asn', 'AGU': 'Ser',     # AxU
    'AUC': 'Ile', 'ACC': 'Thr', 'AAC': 'Asn', 'AGC': 'Ser',     # AxC
    'AUA': 'Ile', 'ACA': 'Thr', 'AAA': 'Lys', 'AGA': 'Arg',     # AxA
    'AUG': 'Met', 'ACG': 'Thr', 'AAG': 'Lys', 'AGG': 'Arg',     # AxG
# G
    'GUU': 'Val', 'GCU': 'Ala', 'GAU': 'Asp', 'GGU': 'Gly',     # GxU
    'GUC': 'Val', 'GCC': 'Ala', 'GAC': 'Asp', 'GGC': 'Gly',     # GxC
    'GUA': 'Val', 'GCA': 'Ala', 'GAA': 'Glu', 'GGA': 'Gly',     # GxA
```

```
    'GUG': 'Val', 'GCG': 'Ala', 'GAG': 'Glu', 'GGG': 'Gly'      # GxG
}
```

Had we used lists, we would have had to write code to search through the list for a codon whose translation we needed. With dictionaries the task is trivial, as you can see in Example 3-4.

Example 3-4. RNA codon lookup from a dictionary

```
def translate_RNA_codon(codon):
    return RNA_codon_table[codon]
```

The nicely formatted, organized dictionary in Example 3-3 was constructed manually. Example 3-5 shows what this dictionary looks like when printed in the interpreter with a 75-column window,‖ as in this book's code examples. This is really just one long line, with "wrapping" at the window edge. Since comments are ignored on input, they aren't part of the output. Notice that the key/value pairs in the dictionary Python constructs from the input appear in an arbitrary order—dictionaries are unordered data structures.

Example 3-5. Dictionary representing the RNA codon table

```
>>> RNA_codon_table
{'CUU': 'Leu', 'UAG': '---', 'ACA': 'Thr', 'AAA': 'Lys', 'AUC': 'Ile', 'AAC
': 'Asn', 'AUA': 'Ile', 'AGG': 'Arg', 'CCU': 'Pro', 'ACU': 'Thr', 'AGC': 'S
er', 'AAG': 'Lys', 'AGA': 'Arg', 'CAU': 'His', 'AAU': 'Asn', 'AUU': 'Ile',
'CUG': 'Leu', 'CUA': 'Leu', 'CUC': 'Leu', 'CAC': 'His', 'UGG': 'Trp', 'CAA'
: 'Gln', 'AGU': 'Ser', 'CCA': 'Pro', 'CCG': 'Pro', 'CCC': 'Pro', 'UAU': 'Ty
r', 'GGU': 'Gly', 'UGU': 'Cys', 'CGA': 'Arg', 'CAG': 'Gln', 'UCU': 'Ser', '
GAU': 'Asp', 'CGG': 'Arg', 'UUU': 'Phe', 'UGC': 'Cys', 'GGG': 'Gly', 'UGA':
 '---', 'GGA': 'Gly', 'UAA': '---', 'ACG': 'Thr', 'UAC': 'Tyr', 'UUC': 'Phe
', 'UCG': 'Ser', 'UUA': 'Leu', 'UUG': 'Leu', 'UCC': 'Ser', 'ACC': 'Thr', 'U
CA': 'Ser', 'GCA': 'Ala', 'GUA': 'Val', 'GCC': 'Ala', 'GUC': 'Val', 'GGC':
'Gly', 'GCG': 'Ala', 'GUG': 'Val', 'GAG': 'Glu', 'GUU': 'Val', 'GCU': 'Ala'
, 'GAC': 'Asp', 'CGU': 'Arg', 'GAA': 'Glu', 'AUG': 'Met', 'CGC': 'Arg'}
```

The way complicated combinations of collections are printed in the interpreter generally does not reveal the structure of the data. To obtain a function that will help you see the structure of your data, you should include the following line in your Python files:

```
from pprint import pprint
```

This imports the function `pprint` from the module `pprint`—short for "pretty-print"—and adds it to your namespace so that you can call it as a function without the module prefix. You can make the function even easier to use if you import it as a shorter alias:

```
from pprint import pprint as pp
```

‖ The output will vary with your window width, which you can change in IDLE by dragging the lower-right corner. On the General tab of IDLE's Preferences window, you can also specify the size for the interpreter window when IDLE starts up.

This imports the function but calls it pp in your namespace. Here is what RNA_codon_table looks like when pretty-printed (to save space, only the first few and last few lines are shown):

```
>>> pp(RNA_codon_table)
{'AAA': 'Lys',
 'AAC': 'Asn',
 'AAG': 'Lys',
 'AAU': 'Asn',
 'ACA': 'Thr',
 # . . . output deleted here
 'UUG': 'Trp',
 'UUT': 'Cys',
 'UUA': 'Leu',
 'UUC': 'Phe',
 'UUG': 'Leu',
 'UUU': 'Phe'}
```

This output goes to the other extreme, printing each dictionary entry on a separate line so that the full output takes up 64 lines. Doing this reveals something interesting that you might not have noticed if several key/value pairs were printed on each line: pprint.pprint orders the dictionary entries alphabetically by their keys. Thus, whereas on input the bases were ordered at each position by the usual TCAG, the order in the output was ACGT. We'll see in the next chapter how to write a function that prints the codon table the way it was organized on input.

Dictionary operations

An interesting ambiguity arises when talking about a value being "in a dictionary." Dictionaries consist of key/value pairs, so are we looking at the keys or the values? Python's answer is "the keys": the membership operators in and not in test whether a key is present; max and min compare keys; and all and any test keys.

The syntax for getting and setting the value associated with a key is simple. Because dictionaries are unordered, there is no equivalent to the flexible slicing operations of sequences. Table 3-14 summarizes the index-based dictionary expressions.

Table 3-14. Dictionary indexing and removal

Expression	Result
d[*key*]	The value associated with *key*; an error is raised if *d* does not contain *key*
d[*key*] = *value*	Associates *value* with *key*, either adding a new key/value pair or, if *key* was already in the dictionary, replacing its value
d[*key*] ·= *value*	Augmented assignment, with · being any of +, -, *, /, //, %, and **; an error is raised if *d* does not have a value for *key* or if the value is not numeric
del *d*[*key*]	Deletes *key* from the dictionary; an error is raised if *d* does not contain *key*

Slice assignment can remove an element from a mutable sequence, so it isn't necessary to use `del` statements with mutable sequences (though they can make your code clearer). Dictionary assignment, however, can only add or replace an element, not remove one. That makes `del` statements more important with dictionaries than with sequences. While there are `dict` methods that remove elements from a dictionary, `del` statements are more concise.

Dictionary methods

Because items in a mapping involve both a key and an associated value, there are methods that deal with keys, values, and key/value pairs. In addition, there are methods that perform the equivalent of the dictionary operations but with more options. There is also a method for adding the key/value pairs from another dictionary, replacing the values of any keys that were already present. Explanations of the details of these operations are found in Table 3-15.

Table 3-15. Dictionary methods

Expression	Result
`d.get(key[, default_value])`	Like `d[key]`, but does not cause an error if *d* does not contain *key*; instead, it returns `default_value`, which, if not provided, is None
`d.setdefault(key[, default_value])`	Like `d[key]` if *key* is in *d*; otherwise, adds *key* with a value of `default_value` to the dictionary and returns `default_value` (if not specified, `default_value` is None)
`d.pop(key[, default_value])`	Like `del d[key]`, but does not cause an error if *d* does not contain *key*; instead, it returns `default_value`, which, if not provided, is None
`d1.update(d2)`	For each key in *d2*, sets `d1[key]` to `d2[key]`, replacing the existing value if there was one
`d.keys()`	Returns a special sequence-like object containing the dictionary's keys
`d.values()`	Returns a special sequence-like object containing the dictionary's values
`d.items()`	Returns a special sequence-like object containing `(key, value)` tuples for the dictionary's keys

Note that the last three methods return "sequence-like objects": they aren't sequences, but they can be used as if they were in many contexts. If you need a dictionary's keys, values, or items in the form of a list, simply call `list` with what the corresponding method returns.

Streams

A *stream* is a *temporally ordered* sequence of *indefinite length*, usually limited to one type of element. Each stream has two ends: a *source* that provides the elements and a *sink* that absorbs the elements. The term "stream" is apt, conjuring as it does the flow

of water into or out of a hose. The more common kinds of stream sources are files, network connections, and the output of a kind of function called a *generator*. Files and network sources are also common kinds of sinks.

Your input to a command-line shell or the Python interpreter becomes a stream of characters (see Figure 3-3). When Python prints to the terminal, that too is a stream of characters. These examples demonstrate the temporal nature of streams: their elements don't necessary exist anywhere other than in the mechanisms that implement the stream itself. For instance, the keystrokes don't "come from" anywhere—they are events that happen in time. (Stream implementations usually involve some kind of *buffer* that does hold onto the items coming in or going out, but that is largely an issue of efficiency and doesn't affect the conceptual interpretation of the stream as a data type.)

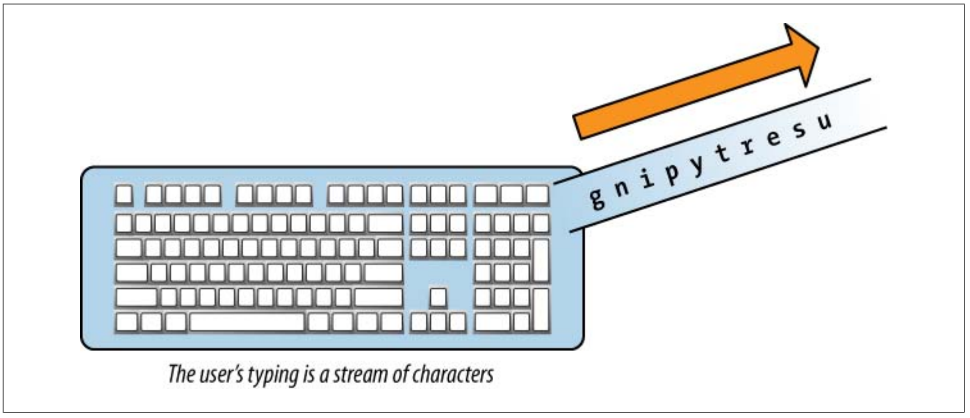


Figure 3-3. A stream of characters issuing from the user's fingers

Files

Usually, the term "file" refers to data stored on media such as a hard disk drive or CD. The term "file" is also used in programming, but with a somewhat different meaning. A Python file is an object that is an *interface* to an external file, not the file itself. File objects provide methods for reading, writing, and managing their instances. The interface is sufficiently natural that the distinction between file objects and the external data they represent can largely be ignored.

The smallest unit of data in external files is, of course, the *bit*, each representing a 1 or a 0. *Bytes* are groups of eight bits. They are generally the basic unit of data that moves through the various levels of hardware, operating system software, and programs. Bytes do get grouped into chunks of various kinds throughout the different levels of computer systems, but programmers can usually ignore the details of the larger groupings.

Python's implementation of file objects manages all the details of moving bytes to and from external files. In most respects, a file object is a kind of sequence. Depending on a parameter supplied when an instance is created, the elements of the file object are either bytes or Unicode characters. Some methods treat files as streams of bytes or characters, and other methods treat them as streams of lines of bytes or characters. Figure 3-4 illustrates an input stream and Figure 3-5 an output stream.

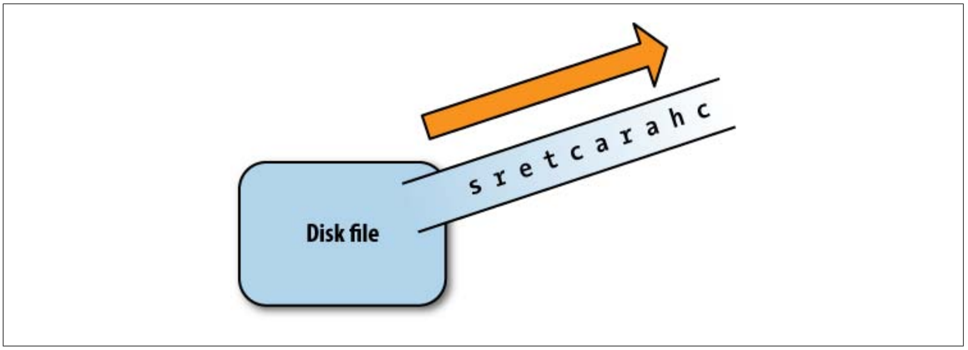


Figure 3-4. Inputting a stream of characters from an external file

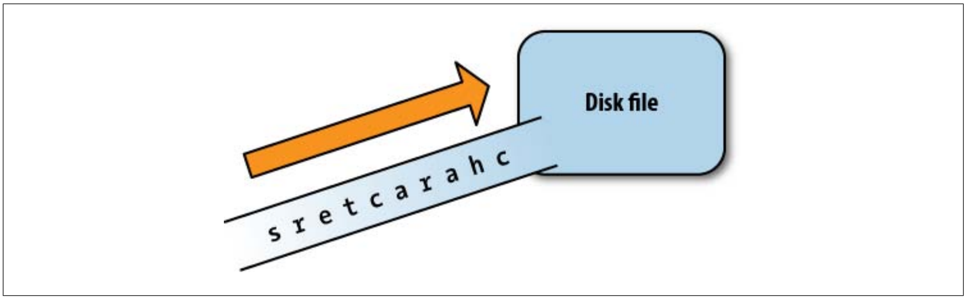


Figure 3-5. Outputting a stream of characters to an external file

Most of the time a file object is a one-way sequence: it can either be read from or written to. It is possible to create a file object that is a two-way stream, though it would be more accurate to say it is a pair of streams—one for reading and one for writing—that just happen to connect to the same external file. Normally when a file object is created, if there was already a file with the same path that file is emptied. File objects can be created to append instead, though, so that data is written to the end of an existing file.

Creating file objects

The built-in function `open(`*`path, mode`*`)` creates a file object representing the external file at the operating system location specified by the string *`path`*. The *`mode`* parameter is a string whose character(s) specify how the file is to be used and its contents interpreted.

The default use is reading, and the default interpretation is text. You can omit the second parameter entirely, in which case the value of *mode* would be `'rt'`.

Specifying `'r'`, `'w'`, or `'a'` without `'t'` or `'b'` is allowable, but the value of `mode` cannot be just `'t'` or `'b'`. If either of those is specified, `'r'`, `'w'`, or `'a'` must be explicitly included. If the string contains a plus sign (`+`) in addition to `'r'`, `'w'`, or `'a'` the file object will be both readable and writable. The order of the characters within the *mode* string is immaterial. Content modes are summarized in Table 3-16 and use the modes listed in Table 3-17.

Table 3-16. Content mode values for opening a file

Value	Mode	Interpretation
t	Text (default)	Characters or strings
b	Binary	Bytes

Table 3-17. Use mode values for opening a file

Value	Initial file position	Read?	Write?
r	Beginning (default)	Yes	No
w	Beginning	No	Yes
a	End	No	Yes
r+	Beginning	Yes	No
w+	Beginning	Yes	Yes
a+	End	Yes	Yes

You could use an assignment statement to name the file object created by `open`, but there's a major problem with this simplistic approach. Due to the way files are implemented both in programming languages and in operating systems, there are various "cleanup" actions that must be performed when a file object is no longer needed. If these actions are not performed after writing with `file` methods, some characters might not get written to the external file. Other problems can also arise, but that's the most important one.

You could call the method `close()` to close a file object when it's no longer needed, but if an error occurs while you're using the file, `close` won't get called, so this won't always work. For this reason Python offers the `with` statement, which combines naming the file object created by `open` with a mechanism to ensure that the appropriate cleanup actions are performed. Even if an error occurs during the execution of the statements in a `with`, the actions necessary to properly close the open file are performed.

STATEMENT

The with Statement

The `with` statement is used to open and name a file, then automatically close the file regardless of whether an error occurs during the execution of its statements. Like a `def` statement, a `with` statement contains an indented block of statements.

```
with open(path, mode) as name:
    statements-using-name
```

More than one file can be opened with the same `with` statement, as when reading from one and writing to the other.

```
with open(path1, mode1) as name1, open(path2, mode2) as name2, ...:
    statements-using-names
```

In versions of Python before 2.6, the executable first line of a file that uses a `with` statement must be:

```
from __future__ import with_statement
```

File methods

Methods for reading from files include the following:

`fileobj``.read([`*`count`*`])`
: Reads *`count`* bytes, or until the end of the file, whichever comes first; if *`count`* is omitted, reads everything until the end of the file. If at the end of the file, returns an empty string. This method treats the file as an input stream of characters.

`fileobj``.readline([`*`count`*`])`
: Reads one line from the file object and returns the entire line, including the end-of-line character; if *`count`* is present, reads at most *`count`* characters. If at the end of the file, returns an empty string. This method treats the file as an input stream of lines.

`fileobj``.readlines()`
: Reads lines of a file object until the end of the file is reached and returns them as a list of strings; this method treats the file as an input stream of lines.

File methods for writing to files include the following:

`fileobj``.write(`*`string`*`)`
: Writes *`string`* to *`fileobj`*, treating it as an output stream of characters.

`fileobj``.writelines(`*`sequence`*`)`
: Writes each element of *`sequence`*, which must all be strings, to *`fileobj`*, treating it as an output stream of lines. Note, however, that although this method's name is intentionally analogous to `readlines`, newline characters are not added to the strings in *`sequence`* when they are written to *`fileobj`*.

Notice that these functions do not provide a location within the sequence to read the characters or lines from or write them to. That's because the location is implicit: the next item in the stream.

In addition, the `print` function has an optional (keyword) argument, `file`, that wasn't documented in Chapter 1. When it's included, `print` writes its output to the value of the `file` argument rather than to the terminal. The `print` function can be thought of as a way to convert an object to a string and then send its characters to an output stream.

Example

To manipulate the contents of a file, it is often convenient to just read the entire contents into a string and then manipulate the string instead. This approach runs into trouble with large files, but we'll see ways to fix that in the next chapter. Meanwhile, we'll look at reading FASTA files.#

The first step will be to read the contents of a FASTA file into a string and split that string at every occurrence of `'>'`. To remind you how `str.split` works, here's a tiny example. I'll emphasize again the importance of building up your functions from small pieces that you can experiment with to see the behavior:

```
>>> '>id1>id2>id3'.split('>')
['', 'id1', 'id2', 'id3']
```

The list returned by `split` in this example has a surprising first element. Imagine how difficult it would be to find the source of bugs or erroneous return values if you were writing a 20-line function to fully process a FASTA file. You're better off starting small, with small functions and small input to test them on. There's nothing wrong—and a lot right—with having functions so small that they each do only one significant thing.

We can capture this bit of code directly in a function that substitutes the contents of a file for the string in the example (Example 3-6). The slice at the end of the function eliminates that surprising and useless first element.

Example 3-6. Reading FASTA sequences from a file, step 1

```
def read_FASTA_strings(filename):
    with open(filename) as file:
        return file.read().split('>')[1:]
```

There are some problems with this function, which we'll fix at the end of this chapter: the description line preceding each sequence is part of the sequence string, and the string contains internal newline characters. We will also see many more examples of ways to read data from FASTA files in later chapters.

#FASTA-formatted files are widely used in bioinformatics. They consist of one or more base or amino acid sequences broken up into lines of reasonable size (typically 70 characters), each preceded by a line beginning with a ">" character. That line is referred to as the sequence's *description*, and it generally contains various identifiers and comments that pertain to the sequence that follows.

Since the call to the `read` method returns the entire contents of the file, we're done with the file after that call. For many situations, though, it's not appropriate to read the entire file. Considerations include:

- The file might be very large, so reading its entire contents could take a long time and use a lot of computer memory.
- The file might contain different sorts of information, each of which needs to be handled differently.
- In many file formats, delimiters separating items aren't as straightforward as they are in FASTA files.

These situations call for using methods that read a restricted amount from the file. The restriction may be in terms of either what it reads—for example, `readline`—or for how long it reads, by providing an argument to specify the number of characters.

Generators

A *generator* is an object that returns values from a series it computes. An example is `random.randint`, as illustrated in Figure 3-6. What's interesting and important about generator objects is that they produce values only on request. Some of the important advantages of generator objects are as follows:

- A generator can produce an infinitely large series of values, as in the case of `random.randint`; its callers can use as many as they want.
- A generator can encapsulate significant computation with the caller requesting values until it finds one that meets some condition; the computation that would have been needed to produce the rest of the possible values is avoided.
- A generator can take the place of a list when the list is so long and/or its values are so large that creating the entire list before processing its elements would use enormous amounts of memory.

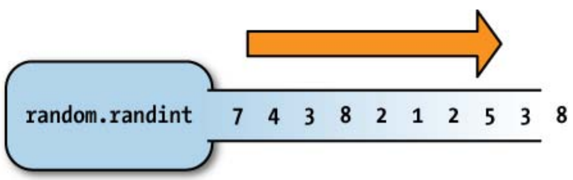


Figure 3-6. A function generating a stream of numbers

The local names of an ordinary function are rebound each time the function is called. The challenge a language faces in implementing generators is that the values of the local names must be retained between calls so that they can be used to compute the next value. (That's how `random.randint` works, for example.) To fully understand how

generators are used, you'll need to learn about some of the statements introduced in the next chapter, but they are mentioned here because of their stream-like behavior.

A value is obtained from a generator by calling the built-in function `next` with the generator object as its argument. The function that produced the generator object resumes its execution until a `yield` statement is encountered. At that point, the value of the `yield` is returned as the value of `next`. The values of parameters and names assigned in the function are retained between calls. In some simple uses, this is the main purpose of using a generator.

STATEMENT

Yielding a Value from a Generator

A function definition that uses a `yield` statement in place of `return` creates a new generator object each time it is called. The generator object encapsulates the function's bindings and code, keeping them together for as long as the object is in use.

```
yield value
```

What happens if `next` is called on a generator when the generator has no more values to produce? The function takes an optional second parameter that specifies the value to return when there are no more values:

`next(`*`generator`*`[, `*`default`*`])`
: Gets the next value from the ***generator*** object; if the generator has no more values to produce, returns ***default***, raising an error if no default value was specified

Collection-Related Expression Features

We've completed our introduction to Python's collection data types. Next, we'll turn to some powerful expression constructs that work with collections.

Comprehensions

A *comprehension* creates a set, list, or dictionary from the results of evaluating an expression for each element of another collection. Each kind of comprehension is written surrounded by the characters used to surround the corresponding type of collection value: brackets for lists, and braces for sets and dictionaries. We'll start with basic list comprehensions, then look at some variations; finally, we'll explore some examples of set and dictionary comprehensions.

The core of a comprehension is an expression evaluated for each element of a collection. It really must be an expression: statements are not permitted. However, in most situations where you want to use a statement in a comprehension, you can call either a function defined for that purpose or an ad hoc lambda expression, discussed a bit later in this chapter. Function calls are expressions, so they can be used in comprehensions. (Yes, function calls, being expressions, can also be used as statements, but they are still expressions.)

List comprehensions

The simplest form of list comprehension is:

```
[expression for item in collection]
```

This produces a new list created by evaluating the expression with each element of the collection. List comprehensions have wide applicability and can often reduce the complexity of code, making it easier to read and write. You'll probably find them strange at first, but once you're comfortable with them you'll see how powerful and convenient they are. Note that the ***`expression`*** must be just that—an expression. Statements are not allowed, but since function calls are expressions they *are* allowed.

Example 3-7 is yet another rewrite of our `validate_base_sequence` function. The expression in this example is `base in valid_bases` and the sequence is `base_sequence.upper()`. (The expression is shown within parentheses to highlight its role, but the parentheses aren't necessary.) The list generated by the comprehension contains only Boolean values. That list is then passed to the built-in function `all` to check whether or not all the values are true.

Example 3-7. Validating base sequences using a list comprehension

```
def validate_base_sequence(base_sequence, RNAflag = False):
    valid_bases = 'UCAG' if RNAflag else 'TCAG'
    return  all([(base in valid_bases)
                for base in base_sequence.upper()])
```

Next, we'll use the functions `random_base` and `random_codon`, which we defined in the previous chapter, to generate a random base sequence. (The function definitions are repeated in Example 3-8 for convenience.) This list comprehension is atypical in that the expression doesn't use the collection items at all—instead, it uses `range` to determine how many times the expression should be evaluated (i.e., how many codons should go in the list). The expression itself is independent of the numbers generated.

Example 3-8. Generating a random list of codons

```
from random import randint

def random_base(RNAflag = False):
    return ('UCAG' if RNAflag else 'TCAG')[randint(0,3)]
```

```
def random_codon(RNAflag = False):
    return random_base(RNAflag) + random_base(RNAflag) + random_base(RNAflag)

def random_codons(minlength = 3, maxlength = 10, RNAflag = False):
    """Generate a random list of codons (RNA if RNAflag, else DNA)
    between minlength and maxlength, inclusive"""
    return [random_codon(RNAflag)
            for n in range(randint(minlength, maxlength))]
```

Often, code makes more sense when it's read from the inside out. Let's take apart the pieces of the last definition to see how it works. We'll assign names to the default values of the parameters so we can evaluate the expressions interactively rather than calling the function:

```
>>> minlength = 3
>>> maxlength = 10
>>> RNAflag = True
```

Next, we name the result of the call `randint`:

```
>>> randnum = randint(minlength, maxlength)
>>> randnum
4
```

Then we show how that can be used as the collection of a list comprehension:

```
>>> [n for n in range(randnum)]
[0, 1, 2, 3]
```

Finally, we make the expression generate a random codon for each of those integers (ignoring the values of the integers):

```
>>> [random_codon(RNAflag) for n in range(randnum)]
['ACT', 'AGG', 'GAG', 'TCT']
```

Of course, since this is all based on a random number, you won't necessarily get the same results if you try these examples yourself.

We can take this example a step further, as shown in Example 3-9. We already have a `translate_RNA_codon` function, which we developed in the section "Dictionaries" on page 67. We can use that to define a function that translates a collection of codons. We could also add the code to the `random_codons` function, but as explained earlier, it is often best to leave small function definitions as they are. You can then use them in the construction of other functions, rather than trying to pack all the desired capabilities into one function.

Example 3-9. Translating random base sequences

```
def random_codons_translation(minlength = 3, maxlength = 10):
    """Generate a random list of codons between minlength and maxlength, inclusive"""
    return [translate_RNA_codon(codon) for codon in
            random_codons(minlength, maxlength, True)]
```

Now let's run a few calls to see what the results look like:

```
>>> random_codons_translation()
['Ser', 'Gly', 'Ile']
>>> random_codons_translation()
['Ser', 'Leu', 'Ala', 'Leu', 'Asn', 'Val', 'Lys', 'Pro', 'His', 'Tyr']
```

Next, let's revisit our primitive FASTA reader. The beginning of each string in the list returned by the definition in Example 3-6 contains information about a base sequence. The rest of the string represents the base sequence itself. Suppose we want to split the description from a base sequence returned from the function `read_FASTA_strings`, as defined in Example 3-6. There happens to be a string method that does just what we need. Given a string ***`string`*** and another string ***`sepr`***, the call ***`string`***`.partition(`***`sepr`***`)` returns a tuple with three elements: the part of ***`string`*** up to the first appearance of ***`sepr`***, ***`sepr`***, and the part of ***`string`*** after ***`sepr`***. Calling `partition` with an argument of `'\n'` will split the description from the base sequence.

We can call `partition` on each string produced by a call to `read_FASTA_strings` and use a list comprehension to collect the results. We could add the necessary code to `read_FASTA_strings`, but we'll adhere to our recommendation of having each function do only one significant thing. Besides, `read_FASTA_strings` is a function that is perfectly good in its own right and that might have uses beyond the current example. Instead, we'll define a new function, `read_FASTA_entries` (Example 3-10).

Example 3-10. Reading FASTA sequences from a file, step 2

```
def read_FASTA_entries(filename):
    return [seq.partition('\n') for seq in read_FASTA_strings(filename)]
```

This function returns a list of the sequences from a FASTA file, each one a tuple of the form `(`***`information`***`, '\n', `***`sequence`***`)`. (Remember to use `pprint.pprint` to help you understand the structure of results that are this complicated.) Next, we need to remove the newline characters from within each sequence string. Again, we'll define a new function: it will use `str.replace` and another list comprehension. We'll take this opportunity to discard the useless `'>'` that begins each description. Example 3-11 shows the new function, `read_FASTA_sequences`.

Example 3-11. Reading FASTA sequences from a file, step 3

```
def read_FASTA_sequences(filename):
    return [[seq[0][1:],                          # ignore the initial '>'
             seq[2].replace('\n', '')]            # delete newlines
            for seq in read_FASTA_entries(filename)]
```

When the value used in the comprehension—`seq` in this case—is a sequence, it can be unpacked into multiple names, which helps make the code clearer. Compare Example 3-12 with Example 3-11. It's not a big difference, but it can be helpful in more complicated examples. Naming the elements of the sequence saves you from having to figure out what the elements were when you come back to the code later.

Example 3-12. Reading FASTA sequences from a file, step 3, unpacked

```
def read_FASTA_sequences(filename):
    return [(info[1:], seq.replace('\n', ''))
            for info, ignore, seq in                    # ignore is ignored (!)
            read_FASTA_entries(filename)]
```

This is probably all beginning to look a bit rich. Play with the code by trying out its bits separately, as demonstrated earlier. You'll have reached a key stage in your Python education when you can look at a function like `read_FASTA_sequences` in Example 3-12 and see more or less immediately what it is doing. In the meantime, it can be very helpful to work through functions like this by entering them into the interpreter line by line and examining the results.

The description lines of FASTA files generally contain vertical bars to separate field values. Example 3-13 defines a new function that calls `read_FASTA_sequences`, then uses `str.split` to return a list of field values for the description instead of just a string.

Example 3-13. Reading FASTA sequences from a file, step 4

```
def read_FASTA_sequences_and_info(filename):
    return [[seq[0].split('|'), seq[1]] for seq in
            read_FASTA_sequences(filename)]
```

Notice how we continue to keep each function small. When building a "vocabulary" of such functions, you don't know which ones you or other programmers will ultimately need. In any case, it's easier to leave the working definitions alone and build on top of them than it is to keep changing already working functions.

Altogether, the functions we developed to read sequences from a FASTA file do the following:

- Split the file contents at `'>'` to get a list of strings representing entries
- Partition the strings to separate the first line from the rest
- Remove the useless `'>'` from the resulting triples
- Remove the newlines from the sequence data
- Split the description line into pieces where vertical bars appear

Each sequence in the result returned is represented by a two-element list. The first element is a list of the segments of the description, and the second is the whole sequence with no newline characters.

What would it look like to do this all at once? A combined definition is shown in Example 3-14.

Example 3-14. Reading FASTA sequences with one compact function

```
def read_FASTA(filename):
    with open(filename) as file:
        return [(part[0].split('|'),
```

```
                 part[2].replace('\n', ''))
                for part in
                [entry.partition('\n')
                 for entry in file.read().split('>')[1:]]]
```

This isn't terrible, but it's near the limit of what you could hope to readily comprehend even once you're comfortable with code like this. You'd probably try to understand it by looking at each piece individually to see what it does, so why not just define each piece as a separate function that names what it does? Doing so makes it easier to read, test, and fix your code.

If ever you do want to produce an all-in-one function like the one in Example 3-14, you should still develop it piece by piece, working from the inside out. Follow the same steps we followed in developing the sequence of separate functions, making one change to the function at a time. It's true that using four separate functions requires a few more lines of code, but they are much more readable than the ensemble of maneuvers in the all-in-one definition. Also, some of them might prove useful in situations that call for less processing of each entry.

Yet another approach is shown in Example 3-15, which defines the function as a series of steps, naming the results of each. This does help to clarify what each piece is doing and can help with debugging, but it's a style more appropriate for programming in older languages. Python's expressiveness and clarity make assignment statements superfluous in many situations. That said, you could write the function using a lot of assignment statements as you are developing. Then, once you've got the definition working, you could remove them and pack everything together into more complex expressions.

Example 3-15. Reading FASTA entries from a file, binding names

```
def read_FASTA(filename):
    with open(filename) as file:
        contents = file.read()                              # only statement inside the with
    entries = contents.split('>')[1:]                       # skip blank first entry
    partitioned_entries = [entry.partition('\n') for entry in entries]
    pairs = [(entry[0], entry[2]) for entry in partitioned_entries]  # omit '>'
    pairs2 = [(pair[0], pair[1].replace('\n', '')) for pair in pairs]
    result = [(pair[0].split('|'), pair[1]) for pair in pairs2]
    return result
```

Finally, there's nothing to prevent you using an intermediary style. For instance, in Example 3-15 there's no reason to have one step that removes the newlines from the second element of the pairs and a second step that splits the first element at vertical bars. These actions could be merged into one step:

```
[(pair[0].split('|'), pair[1].replace('\n', '')) for pair in pairs]
```

Set and dictionary comprehensions

Set comprehensions work like list comprehensions. They are enclosed in braces and produce a set:

```
{expression for item in collection}
```

One important difference from list comprehensions is that the result of a set comprehension doesn't necessarily have as many elements as the collection used by the comprehension. Because it's a set, duplicates accumulated during the comprehension are ignored. Also, the value of ***expression*** cannot be an instance of a mutable built-in collection, because that could not be an element of a set.

Dictionary comprehensions are also enclosed in braces, but they are a bit more complicated because they require key/value pairs:

```
{key-expression: value-expression for key, value in collection}
```

Just as when you call `dict` with a collection argument, the collection in a dictionary comprehension must contain only two-element lists or tuples. As always with dictionaries, the key values may not be mutable collections.

Taking a list of pairs of the form (***info-fields***, ***sequence***) as produced by `read_FASTA` in Example 3-16 and assuming the first info field is `'gi'` and the second the identifier, it's easy to construct a dictionary of sequences indexed by identifier.

Example 3-16. Constructing a dictionary with a comprehension

```
def make_indexed_sequence_dictionary(filename):
    return {info[0]: seq for info, seq in read_FASTA(filename)}
```

Generator expressions

A bit earlier in this chapter we discussed using the `yield` statement to create a function that returns a new generator object each time it is called. Comprehensions are often a more convenient way to express computations that otherwise would be defined as separate functions, and Python has something similar for generators. A *generator expression* is syntactically like a list or set comprehension, except that it is surrounded with parentheses and its value is a generator:

```
(expression for item in collection)
```

Since `yield` is implicit in generator expressions, they are often quite a bit simpler than equivalent generator-producing functions.

Generators look just like equivalent list or set comprehensions. Accessing individual elements of these three collection types requires following different rules, though:

- There is no way to access an individual element of a set.
- There is only one element that can be accessed from a generator, and that is obtained by calling `next`.
- Any element of a list can be directly accessed with an index expression.

Some collection types can accept any other kind of collection as their argument when called as a function. Thus, `set(g)` will return a set constructed from all the elements of the generator *g*, and `list(g)` will return a list of all the elements of the generator *g*. Note the similarity to files, another kind of stream: `next` and `list` for generators correspond to `readline` and `readlines` for files. You can get the next item of the stream, or get them all. However, you cannot ask for a specific item using an index expression.

Generator expressions are useful when the expressions they encompass require substantial computational resources or the collections are large. By using `next` to get one element of the generated stream at a time and stopping when certain criteria have been fulfilled, we can avoid unnecessary computation. Also, like generator functions, generator expressions can generate infinite streams. (But don't try to call `set`, `list`, or `tuple` on one of those!) Example 3-17 shows how a generator expression could be used to produce amino acid abbreviations from an RNA sequence.

Example 3-17. Generating amino acid translations of codons

```
def aa_generator(rnaseq):
    """Return a generator object that produces an amino acid by translating
    the next three characters of rnaseq each time next is called on it"""
    return (translate_RNA_codon(rnaseq[n:n+3])
            for n in range(0, len(rnaseq), 3))
```

Since we're ignoring the problem of running out of codons, the definition does not handle the extra base or two that might appear at the end of the string following the last codon. Here's how this works:

```
>>> aagen = aa_generator('CCACCGCACCAACAGCGC')
>>> next(aagen, None)
'Pro'
>>> next(aagen, None)
'Pro'
>>> next(aagen, None)
'His'
>>> next(aagen, None)
'Gln'
>>> next(aagen, None)
'Gln'
>>> next(aagen, None)
'Arg'
```

It is an error to call `next` on a generator that has no more elements to produce, unless a second argument is included in the call. In that case, the second argument is a value to return when the generator is exhausted. It is *not* an error to call `next` on an exhausted generator as long as the second argument is included. Consider the following two calls after the preceding series of inputs:

```
>>> next(aagen, None)                  # OK, default value None returned
>>> next(aagen)                        # Error, since no default value is provided
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`StopIteration` doesn't look much like the name of an error message, and we haven't even discussed "iteration" yet (we will in the next chapter). Actually, `StopIteration` is used in the implementation of certain parts of Python, and we'll even use it in some later examples. It's not so much an error as an indication that the end of a stream has been reached. What makes it an error in this case is that the call to `next` didn't say what to do when the stream was exhausted.

Conditional comprehensions

The value of a comprehension does not necessarily have to include an element corresponding to every element in its collection. One or more tests can be added to determine the elements for which the comprehension expression will be evaluated. This is called *filtering* and is implemented with a *conditional comprehension*:

```
[expression for element in collection if test]
```

Multiple tests can be included as a series of "if" components of the comprehension. Example 3-18 demonstrates the use of a conditional comprehension. The function filters the names returned by `dir` to eliminate names beginning with underscores.

Example 3-18. Filtering out the underscore names from dir

```
def dr(name):
    """Return the result of dir(name), omitting any names beginning with an underscore"""
    return [nm for nm in dir(name) if nm[0] != '_']
```

Files often contain different kinds of information on different lines, such as the descriptions and sequences of a FASTA file. This kind of situation is another good candidate for a conditional comprehension. Example 3-19 shows how easy it is to extract just sequence descriptions from a FASTA file and split them into fields at their vertical bars.

Example 3-19. Reading FASTA descriptions from a file

```
def get_FASTA_descriptions(filename):
    with open(filename) as file:
        return [line[1:].split('|') for line in file if line[0] == '>']
```

Conditional comprehensions work the same way for sets as they do for lists. We could use one to find all the codes in the third position of the description lines of a FASTA file. For example, a file obtained from GenBank will begin with `gi`, an identifier, `gb`, and a GenBank accession number. However, the third field of a FASTA file obtained from another source or generated by an application might have a different meaning. Example 3-20 defines a function that looks at the descriptions in a FASTA file and returns a list of all the different codes it finds in the third field.

Example 3-20. Reading FASTA descriptions using set comprehension

```
def get_FASTA_codes(filename):
    with open(filename) as file:
```

```
    if len(line.split('|')) < 3:
        return []
    return {line.split('|')[2] for line in file if line[0] == '>'}
```

Conditional dictionary comprehensions are similar. They use braces and must supply both a key and a value, separated by a colon. In Example 3-20 no provision was made for entries whose descriptions begin with something other than `'gi'`. The definition presented in Example 3-21 omits such sequences (which may or may not be a useful thing to do).

Example 3-21. Constructing a selective dictionary

```
def make_gi_indexed_sequence_dictionary(filename):
    return {info[1]: seq for info, seq in read_FASTA(filename)
            if len(info) >= 2 and info[0] == 'gi'}
```

Even generator expressions can include conditionals. They produce values only for those elements of the collection for which the conditional(s) are true. Example 3-22 shows an interesting use of a conditional generator expression to find the first element in one list that is also in a second.

Example 3-22. Using a generator to find the first common element

```
def first_common(list1, list2):
    """Return the first element in list1 that is in list2"""
    return next((item for item in list1 if item in list2), None)
```

Nested comprehensions

Comprehensions can have more than one `for` in them. When they do, the innermost ranges over its collection for each of the next one out's collection, and so on. It is rare to see more than two `for` sections in a comprehension, but occasionally three can be useful.

Example 3-23 shows how clear a terse comprehension can be while defining a significant computation. The function returns a list containing every three-character combination of the (unique) characters in its argument. It first converts its argument to a set in order to eliminate any duplicates. The intended use is to generate a list of the strings representing all the codons for a given collection of bases. The default is `'TCAG'`, but it could be called with `'UCAG'`.

Example 3-23. A nested comprehension for generating codons

```
def generate_triples(chars='TCAG'):
    """Return a list of all three-character combinations of unique characters in chars"""
    chars = set(chars)
    return [b1 + b2 + b3 for b1 in chars for b2 in chars for b3 in chars]
```

In fact, the function is entirely general. It will produce a list containing the result of evaluating the `+` operator with every combination of three values in its argument:

- A string of any length:

```
>>> generate_triples('01')
['111', '110', '101', '100', '011', '010', '001', '000']
```

- Any nonstring collection containing one-character strings:

```
>>> generate_triples(('.', '-'))
['---', '--.', '-.-', '-..', '.--', '.-.', '..-', '...']
```

- Any nonstring collection containing strings of any length:

```
>>> generate_triples({'0', '555'})
['555555555', '5555550', '5550555', '55500', '0555555', '05550', '00555', '000']
```

- Any nonstring collection containing anything that supports `+`:

```
>>> generate_triples(range(2,4))
[6, 7, 7, 8, 7, 8, 8, 9]
```

Functional Parameters

Earlier in this chapter we saw two functions and one method with optional `key` parameters. How this works requires some discussion.

The parameter "key"

We first encountered the two functions, `max` and `min`, in Chapter 1, where functions were introduced. There they were described as taking any number of positional arguments, but these two functions can also be called with a collection as the only positional argument. (That wasn't mentioned before because collections hadn't yet been introduced.) Both forms accept an optional `key` parameter, which is expected to be a function. The value of the `key` argument is called on each element of the collection. The values returned by those calls are used to select the maximum or minimum element:

```
>>> max(range(3, 7), key=abs)
7
>>> max(range(-7, 3), key=abs)
-7
```

Note that although `abs` was used here to compare the elements of the range, what's returned is the *element* for which `abs` returned the largest value—not the *value* of calling the key function on the element. For a more interesting illustration, consider a list `seq_list` containing RNA base sequences. We could select the sequence with the lowest GC content by calling `min` with `key = gc_content` (a function defined in the previous chapter).

Earlier in this chapter, the method `list.sort` was described. It too can take an optional `key` parameter. The `key` parameter works the way it does for `min` and `max`—the value of `key` is called on each element, and the elements of the list are sorted according to the values returned. The `sort` method uses `<` to order the elements of the list. If a call does not include a `key` argument, the elements of the collection being sorted must have

mutually comparable types: for example, a list of numbers or a list of strings, but not a mixed list of numbers and strings. If the call does include a key argument, its value is called for each element and the results are compared using <. Let's look at an example:

```
>>> lst = ['T', 'G', 'A', 'G', 't', 'g', 'a', 'g']
>>> lst
['T', 'G', 'A', 'G', 't', 'g', 'a', 'g']
>>> lst.sort()
>>> lst
['A', 'G', 'G', 'T', 'a', 'g', 'g', 't']
```

> Sorting lst in this example did change the order of its elements, but in a way that may surprise you: all uppercase letters are considered "less than" all lowercase letters. This behavior has a long history in computer technology, the details of which are not important; keep it in mind, though, when sorting lists of strings.

The way to have a list sorted according to an order you define is to provide a key function. In this example we can eliminate the distinction between upper- and lowercase letters by providing a key that returns the uppercase (or, equivalently, the lowercase) version of its string argument.

> **Python 2:** The key argument is a function of two arguments, defaulting to cmp, that is called to compare two elements of the collection.

```
>>> lst.sort(key=str.lower)
>>> lst
['A', 'a', 'C', 'c', 'G', 'g', 'T', 't']
>>>
```

> Another thing you might have noticed is that after sorting using str.lower, each uppercase letter appears before the corresponding lowercase one. This is not due to anything about letter case; for technical reasons it is helpful for sorting functions to be *stable*, meaning that if two items are equal, they will be in the same order after the sort as they were before it.

One use of sorting by key is to order a list of base sequence strings where some use lowercase and some uppercase (perhaps because they were obtained from different sources, sites, databases, or programs). The default sort does not do what we want:

```
>>> seqs = ['TACCTATACCGGCTA', 'cacctctaccgta', 'AACCTGTCCGGCTA']
>>> seqs.sort()
>>> seqs
['AACCTGTCCGGCTA', 'TACCTATACCGGCTA', 'cacctctaccgta']
```

The sequence beginning with `'c'` should go between the other two but appears at the end. This is only a small example; imagine what would happen when sorting a large number of sequences, some uppercase and others lowercase (or worse, using mixed case). What we want instead is for `sort` to ignore case:

```
>>> seqs = ['TACCTATACCGGCTA', 'cacctctaccgta', 'AACCTGTCCGGCTA']
>>> seqs.sort(key=str.lower)
>>> seqs
['AACCTGTCCGGCTA', 'cacctctaccgta', 'TACCTATACCGGCTA']
>>>
```

Function objects

The value of the `key` argument in the preceding examples appeared to be just a name. However, as with any use of names in expressions, it is the value the name is bound to that is actually passed to the function. The value of `abs` is a built-in function, and the value of `str.lower` is a method in the class `str`.

The idea of using a function or method as an argument to another function or method may seem strange, but functions and methods are objects, just like everything else in Python. The `def` statement creates a function object and names it.

You can type the name of a function in the interpreter to see what kind of object it is (what is printed for built-in functions and methods is somewhat different than what you'll see for ones you define or import, and what is printed for functions differs from what is printed for methods, but the values are all objects):

```
>>> max
<built-in function max>
>>> gc_content
<function gc_content at 0x74a198>
>>> str.lower
<method 'lower' of 'str' objects>
>>> lower
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lower' is not defined
>>>
```

The reason for the error in the last example is that there is no function called "lower". Only the `str` class provides a `lower` method, so either `str` or an instance of `str` must appear before `lower`, with a dot in between.

Like any value, a function can have multiple names. Binding a name to an existing function is no different than binding a name to an existing object of any other type:

```
>>> fn = gc_content
>>> fn('ATCCCGGG')
0.75
```

A function call is a kind of expression. In evaluating the expression Python obtains the function object named in the call, binds its parameters to the arguments in the call, and

begins executing the function. What "route" (name) it takes to get to the object has no effect on the process.

Anonymous functions

Passing functions as arguments is not as esoteric as it may seem. There are many situations in which a functional parameter avoids a great deal of repetitive code. Imagine if you had to define a separate function for every different way you might want to find the maximum value in a collection or sort a list. Just as other parameters generalize pieces of computation, so do functional parameters.

Functional parameters are powerful tools, but you still have to write the different functions to pass as arguments. Functions passed as arguments are generally small and simple. Defining a lot of these little functions used solely to pass to other functions gets tedious. Also, definitions used in a subsidiary way like these really shouldn't be as prominent within a file as normal functions—they'd distract the reader and obscure the real structure of the file's code.

Fortunately, Python has a mechanism for defining lightweight functions without using `def`. These functions don't even have names—they are *anonymous*. Although they are functions, they are defined by an expression, not a statement. This kind of expression is called a *lambda expression*.* Its syntax is as follows:

```
lambda args: expression-using-args
```

A lambda expression has one or more arguments. The result is an anonymous function that can be used anywhere you would use a named function. The `key` parameters of `min`, `max`, and `lst.sort` expect functions of one argument, so a lambda passed to one of these must have exactly one argument. The expression can be any kind of expression, including one with function calls, but it must be an expression, not a statement.

Lambda expressions can be passed as arguments, just as if they were named functions. They have a few important benefits, though: they are a little more concise than functions defined with `def`; they are anonymous, so there's no risk of them being used inappropriately or accidentally elsewhere in your code; and they keep the definition right where it's used, which improves readability. Actually, you can name a lambda expression with an assignment statement, just as you'd name any other value. The effect would be the same as if you had used `def` to define and name the function. The following two ways of defining the function named `fn` are in all ways equivalent:

```
def fn (x, y):
    return x*x + y*y

fn = lambda x, y: x*x + y*y
```

In fact, when a lambda expression is passed to another function, it *does* get named. Just like any value passed as an argument in a function call, the corresponding name in the

* The term "lambda" and the corresponding concept come from formal logic and mathematics.

function's parameter list is bound to the lambda expression. Inside the function definition, there is no difference whatsoever between using a parameter name to call a function that has been defined with `def` or an anonymous function defined by a lambda expression.

Let's look at an example. Python has two built-in functions called `all` and `any`. They each take an iterable as an argument. The function `all` returns `True` if *all* the elements of the iterable are true, while `any` returns `True` if *any* of them are. Arguably, it might have been better for those functions to have been defined to take an optional function to call on each element and then to check the result returned by the function instead of testing the elements themselves. If no function were provided, the default would be to use the element itself.

We can define functions to do this. It is never a good idea to redefine built-in functions, so we must invent two other names: `every` and `some`, perhaps. Example 3-24 shows a definition for `some` that uses a conditional generator. The function calls `next` once. If there is an ***`item`*** in ***`coll`*** for which `pred`(*`item`*) is true, the generator produces at least one `True` value. If, however, `pred`(*`item`*) is not true of any ***`item`***, the generator produces an empty stream and `next` will use the default value `False`, as specified in its call.

Example 3-24. Definition of a function with a functional argument

```
def some(coll, pred=lambda x: x):
    """Return true if pred(item) is true for some item in coll"""
    return next((True for item in coll if pred(item)), False)

>>> some([0, '', False])
True
>>> some([0, '', False])
False
>>> some([0, '', 4])
True
>>> some([0, 1, 4], lambda x: isinstance(x, int) and x % 2)         # odd ints?
True
>>> def odd(n):
...     return bool(n % 2)
)>>> some([0, 2, 4], odd)
False
>>> some({'a string', ''}, len)
False
>>> some({tuple(), ''}, len)
False
```

The last two lines of input illustrate that `some` can be used on any collection, not just a sequence. The definition of `odd` and the input before and after it show that the name of a function and a lambda expression have the same effect in a function call.

> It is rarely necessary to call `bool`: anywhere you would use the result of an expression on which you can call `bool`, you can just use the expression by itself. The definitions of `some` and `odd` in this example did use `bool` because the functions are predicates like `all` and `any`; thus, they are expected to always return a Boolean value.

Returning to the example of sorting a list of strings in mixed case, suppose we want to order them by size first, and then alphabetically. We could order them by size easily enough by providing `len` as the `key` argument, just as we ordered them alphabetically after converting all of the sequences to lowercase. However, there's no straightforward way to order them by multiple criteria without passing a function defined for that purpose as the `key` argument.

What would such a function look like? It could be anything that returns a result appropriate for use in a `<` expression. It turns out that the definition of `<` when its operands are sequences is to compare their first elements, then their second, and so on, until one is "less than" the other. For example:

```
>>> l = [(3, 'abc'), (5, 'ghijk'), (5, 'abcde'), (2, 'bd')]
>>> l.sort()
>>> l
[(2, 'bd'), (3, 'abc'), (5, 'abcde'), (5, 'ghijk')]
```

To sort by both length and alphabetically ignoring case, we need a function that returns a pair (tuple or list) whose first element is the sequence's length and its second is the sequence converted to either lower- or uppercase. We could define such a function:

```
def sequence_sort_key(seq):
    return (len(seq), seq.lower())
```

Then `l.sort(key=sequence_sortkey)` would do the trick. We probably don't need such a function anywhere else, though, and it's better to use a lambda expression than define a superfluous function:

```
>>> l.sort(key=lambda seq: (len(seq), seq.lower())))
```

It's never *necessary* to use a lambda expression. Once you become familiar with them, though, you'll find them a welcome tool. Even if you don't use them, learning about them will help you better understand the true nature of functions. We'll be using lambda expressions for some important purposes later in the book.

Tips, Traps, and Tracebacks

Tips

- When typing the contents of a collection inside its enclosing punctuation—parentheses for tuples, brackets for lists, and curly braces for sets and

dictionaries—it's a good idea to always end with an extra comma. That avoids errors if you later add more elements and forget to add the comma after what had previously been the last element. Then, when you do add more elements, make sure you include a comma after the new ones too.

- Keep in mind that there only a few functions that are used with string arguments. Most string operations are method calls; for instance, `'ATTCATT'.count('A')`.

Language features

- Don't call `close` on files opened using a `with` statement; the file will automatically be closed at the end of the statement's execution.
- Practice using comprehensions. They may appear strange at first, but they are an elegant and powerful way to express computations that construct one collection from another, and these are quite common.
- Give generator expressions and generator functions a try.
- Practice using lambda expressions for anonymous function definitions that consist of only an expression.
- Where does Python find files whose names are given as arguments to `open`? If the argument is an absolute path—i.e., one beginning with a slash—that is the path to the file. Otherwise, the path is relevant to the "current" directory. Determining which directory is current can be confusing. Some of the ways Python determines the current directory are:
  - When the interpreter is started from a command line, the current directory is the directory that was current in the command-line environment.
  - The directory that is current when IDLE starts is platform-dependent: on OS X, it is the user's *Documents* folder; on Linux, in might be the *home* directory or the *home/Documents* directory; and on Windows, it is the Python installation directory (although, as with all Windows programs, an alias to IDLE can be set up and given a different start directory through its property dialog).
  - When a file is run from IDLE, the current directory is changed to the directory of the file.
- You can find out what the current directory is by typing the following to the interpreter:

```
import os
os.getcwd()
```

Developing and testing code

- Build programs step by step, keeping function definitions small and testing your code each time you add one or two new ones.
- If you test your code after every change, or every few changes, you'll have a much better idea of what caused a new error than if you write a whole bunch of new code

and try to make it work. Ideally, you should test every new function definition immediately, then test how other functions work when you add calls to the new function to them.

- Use assertion statements liberally to:

  — Validate arguments.

  — Validate expected partial results in the middle of function definitions.

  — Validate results obtained from calls to one of your functions from another.

  Why not just use calls to `print`? There are several reasons:

  — It can be quite a strain—and quite boring—to repeatedly examine similar-looking output as you write and test your code.

  — It is very easy to overlook small problems in the output.

  — The details of formatting output sufficiently to be able to examine it are often needed only for that purpose, in which case it is a waste of time—let Python do the checking with `assert` statements.

  Very common tests to put in assertions include:

  — The expected length of a collection.

  — The range of a numerical value returned from a call to one of your functions (e.g., `gc_content`).

  — The contents of lines read from a file.

- When you do look at collection values—especially as you begin working with collections that contain other collections—remember to use `pprint.pprint`. You might as well just put this line:

  ```
  from pprint import pprint as pp
  ```

  at the beginning of every file to make it easier to use.

- Periodically step back from the details of the code you are developing and look at its overall "shape." Keep an eye out for long functions, repeated code, strings that appear in more than one place, and so on; these are candidates for new definitions and assignments.

Traps

- `(value)` is simply `value` with parentheses around it, so it is equal to `value`, not a one-element tuple. A one-element tuple is written with a trailing comma: e.g., `(value,)`.
- An augmented assignment `list1 += list2` *modifies* `list1`, whereas `list1 + list2` creates a new list.
- The result of `file.readline` includes a newline character; the lines returned by `file.readlines` do not.

- In a comprehension that just selects elements of a collection, it is very easy to forget to include the `for` part, since a comprehension like this looks perfectly natural:

  ```
  [x in lst if x > 0]
  ```

  It should be:

  ```
  [x for x in lst if x > 0]
  ```

- The result returned by `max` or `min` is an element of its collection argument, *even if a* `keyfn` *argument is provided*. The `keyfn` is used to compute a value for the purpose of comparison; when `max` or `min` identifies the appropriate element it returns that element, not the result of calling `keyfn` on that element.
- The `keys`, `values`, and `items` methods of `dict` look like they should return lists, but they don't; they return special types of objects, which you will see if you call one of them on a dictionary. These objects are iterables, but they are not in themselves equal to any list. Usually the only thing you do with the results of these methods is use them in a `for` statement or comprehension. If you do need a list value, call `list` on the result of the method call, but keep in mind that the order within the list is not predictable.
- Your computer's operating system may be suppressing common file extensions in file browser windows. This can lead to all sorts of problems because you are not seeing the actual name of the filename. Turn off this consumer-oriented feature, as follows:

  OS X
  : In the Finder Preferences, click Advanced (the gear icon) at the top and turn *on* "Show all File Extensions."

  Windows
  : In a Windows Explorer window, select "Folder Options" from the Tools menu, click the View tab, and uncheck "Hide extensions for known file types."

  Unix/Linux
  : Different desktop environments may do this differently, but you should be able to figure out how to make sure file extensions are always shown in the standard file browser.

- To open a file that contains Unicode characters, it is necessary to include `encoding='utf-8'` as the third argument to `open`.

Tracebacks

Representative error messages include:

`NameError: [len(x) in lst]`
: Missing `for x` before `in`.

`SyntaxError: [len(x) in lst if x > 3]`
: Missing `for x` before `in`.

`IndexError: string index out of range`
: For a string of length `N`, an index (i.e., the value between square brackets) must be in the range `-N <= index < N-1`.

`TypeError: unhashable type 'list'`
: `TypeError`s have many causes and variations. “Unhashable” indicates a violation of the prohibition against using mutable built-in types as elements of sets or keys of dictionaries.

`object is not iterable: 'builtin_function_or_method'`
: A function name was used in place of a function call in the `in` part of a comprehension; i.e., you forgot the parentheses! This is a common mistake when calling `dict.keys()`, `dict.values()`, and `dict.items()`.

CHAPTER 4

Control Statements

> This chapter's material is rich and intellectually challenging. Don't give up if you start to feel lost (but do review it later to make sure you have absorbed it all). This chapter, together with the next, will complete our introduction to Python. To help you understand its contents, the chapter ends with some extended examples that reiterate the points made in its shorter examples. The rest of the book has a very different flavor.

Chapters 1 and 2 introduced *simple statements*:

- Expressions, including function calls
- Assignments
- Augmented assignments
- Various forms of import
- Assertions
- `return`
- `yield` (to implement generators)
- `pass`

They also introduced the statements **`def`**, for defining functions, and **`with`**, to use with files.[*] These are *compound statements* because they require at least one indented statement after the first line. This chapter introduces other compound statements. As with **`def`** and **`with`**, the first line of every compound statement must end with a colon and be followed by at least one statement indented relative to it. Unlike **`def`** and **`with`** statements, though, the other compound statements do not name anything. Rather, they determine the order in which other statements are executed. That order is traditionally

[*] The **`with`** statement is more general than how it was described in Chapter 2: it actually does not need to name the object of the **`with`** in an **`as`** portion of the statement, and its use is not limited to files. However, the way it was described is the only way it is used in this book.

called the *control flow* or *flow of control*, and statements that affect it are called *control statements*.[†]

Some kinds of compound statements can or must have more than one *clause*. The first line of each clause of a compound statement—its *header* in Python terminology—is at the same level of indentation as the headers of the statement's other clauses. Each header begins with a keyword and ends with a colon. The rest of the clause—its *suite*—is a series of statements indented one level more than its header.

> The term "suite" comes from Python's technical documentation. We'll generally use the more common term *block* instead. Also, when discussing compound statements, we frequently refer to clauses by the keywords that introduce them (for example, "a `with` clause").

Figure 4-1 illustrates the structure of a multi-clause compound statement. Not all compound statements are multi-clause, but every clause has a header and a suite containing at least one statement (if only `pass`).

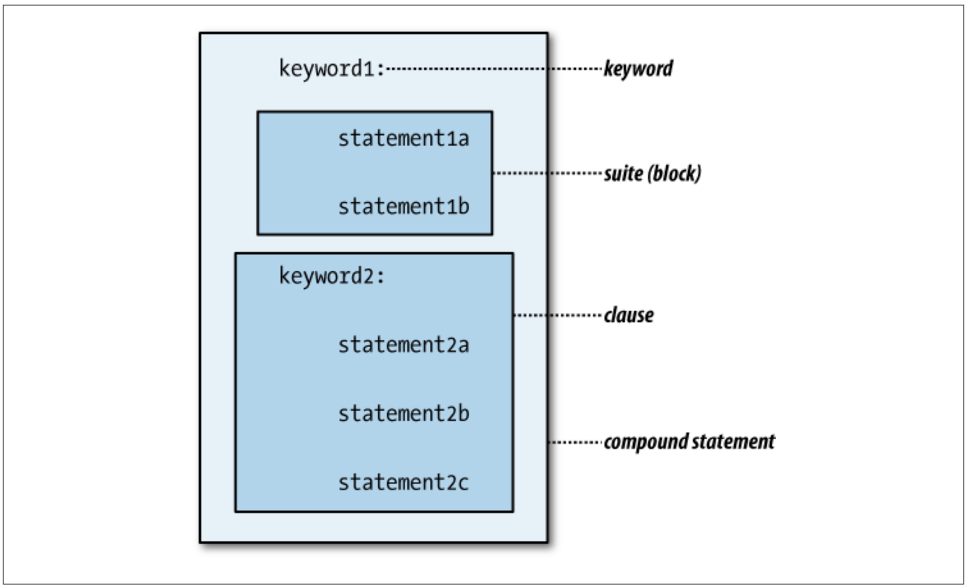


Figure 4-1. The structure of compound statements

† Strictly speaking, a function call expression (not a function definition) also affects the flow of control since it causes execution to proceed in the body of the function. Function calls can appear as separate statements or as part of expressions; either way, execution of the statements that follow or the rest of the expression is suspended until the function returns. From the point of view of the code calling the function, it is a single step that does not affect the order in which statements are executed.

The statements discussed in this chapter are the scaffolding on which you will build your programs. Without them you are limited to the sort of simple computations shown in the examples of the previous chapters. The four kinds of compound statements introduced here are:

- Conditionals
- Loops
- Iterations
- Exception handlers

> Starting later in this chapter, some common usage patterns of Python functions, methods, and statements will be presented as abstract *templates*, along with examples that illustrate them. These templates are a device for demonstrating and summarizing how these constructs work while avoiding a lot of written description. They contain a mixture of Python names, self-descriptive "roles" to be replaced by real code, and occasionally some "pseudocode" that describes a part of the template in English rather than actual Python code.
>
> The templates are in no way part of the Python language. In addition to introducing new programming constructs and techniques as you read, the templates are designed to serve as references while you work on later parts of the book and program in Python afterwards. Some of them are quite sophisticated, so it would be worth reviewing them periodically.

Conditionals

The most direct way to affect the flow of control is with a *conditional statement*. Conditionals in Python are compound statements beginning with `if`.

STATEMENT

Simple Conditional

The basic form of a conditional statement controls whether or not a block of statements will get executed.

```
if expression:
    statements
```

If the expression is true, the statements are executed; otherwise, they are skipped. This is like the `if` inside a conditional comprehension, but it's more powerful since it is followed by statements, whereas a comprehension is limited to expressions.

During the import of a module `__name__` is bound to the name of the module, but while the file is being executed `__name__` is bound to `'__main__'`. This gives you a way to include statements in your Python files that are executed only when the module is run or, conversely, only when it is imported. The comparison of `__name__` to `'__main__'` would almost always be done in a conditional statement and placed at the end of the file.

A common use for this comparison is to run tests when the module is executed, but not when it is imported for use by other code. Suppose you have a function called `do_tests` that contains lots of assignment and assertion statements that you don't want to run when the module is imported in normal use, but you do want to execute when the module is executed. While informal, this is a useful technique for testing modules you write. At the end of the file you would write:

```
if __name__ == '__main__':
    do_tests()
```

There are several forms of conditional statements. The next one expresses a choice between two groups of statements and has two clauses, an `if` and an `else`.

STATEMENT

One-Alternative Conditional

In this form of conditional, the statements in the first block are executed if the expression is true; otherwise, the statements in the second block are executed.

```
if expression:
    statements1
else:
    statements2
```

A simple use of the one-alternative form of conditional is to expand the test for whether a file is being imported as opposed to being executed. We can set it up so that one thing happens when the file is imported and a different thing happens when it's executed.

This example shows only one statement in each block. There could be others, but another reason to group statements into simple functions is so you can invoke them "manually" in the interpreter during development and testing. You might run `do_tests`, fix a few things, then run it again. These test functions are useful whether invoked automatically or manually:

```
if __name__ == '__main__':
    do_tests()
else:
    print(__name__, 'has been imported.')
```

The third form of conditional statement contains more than one test. Except for the `if` at the beginning, all the test clauses are introduced by the keyword `elif`.

STATEMENT

Multi-Test Conditional

This form of the statement contains an `if` clause and any number of `elif` clauses. The statement may end with an `else` clause, but that is not necessary.

The expressions in the `if` clause and each of the `elif` clauses are evaluated in order until one is true. Then, that clause's statements are executed and the rest of the conditional is skipped. If none of the expressions is true and there is an `else` clause, its statements get executed; otherwise, none of the statements are executed.

```
if expression1:
    statements1
elif expression2:
    statements2
# . . . any number of additional elif clauses
else:
    statements
```

Python has a rich repertoire of mechanisms for controlling execution. Many kinds of maneuvers that would have been handled in older languages by conditionals—and could still be in Python—are better expressed using these other mechanisms. In particular, because they emphasize *values* rather than *actions*, conditional expressions or conditional comprehensions are often more appropriate than conditional statements.

Programming languages have many of the same properties as ordinary human languages. Criteria for clear writing are similar in both types of language. You want what you write to be:

- Succinct
- Clear
- Accurate

> It's important not to burden readers of your code (you included!) with too many details. People can pay attention to only a few things at once. Conditionals are a rather heavy-handed form of code that puts significant cognitive strain on the reader. With a little experience and experimentation, you should find that you don't often need them. There will be examples of appropriate uses in the rest of this chapter, as well as in later ones. You should observe and absorb the style they suggest.

Loops

A *loop* is a block of statements that gets executed as long as some condition is true. Loops are expressed in Python using `while` statements.

> **STATEMENT**
>
> ### Loop
>
> The basic form of loop begins with the keyword `while` and an expression.
>
> ```
> while expression:
>     statements
> ```
>
> If the expression is true, the statements are executed and the expression is evaluated again. As long as the expression is true, the statements are executed repeatedly. Once the expression is false, the statements are skipped, completing the execution of the `while` statement.

Note that the test may well be false the first time it is evaluated. In that case, the statements of the block won't get executed at all. If you want some code to execute once the test is false, include an `else` clause in your loop.

> **STATEMENT**
>
> ### Loop with Final Clause
>
> This form of loop statement adds an `else` clause whose statements are executed after the expression evaluates to false.
>
> ```
> while expression:
>     statements1
> else:xs
>     statements2
> ```

There are two simple statements that are associated with both loops and iterations (the subject of the next section): `continue` and `break`.

> **STATEMENT**
>
> ### Loop Interruption
>
> Occasionally it is useful to interrupt the execution of a loop's statements. The `continue` statement causes execution of the loop to return to the test. The `break` statement interrupts the entire loop's execution, causing the program to continue with the statement following the `while`.

The `continue` statement is rarely used in Python programming, but it's worth mentioning it here in case you run across it while reading someone else's Python code. The `break` statement is seen somewhat more often, but in most cases it is better to embed in the loop's test all the conditions that determine whether it should continue rather than using `break`. Furthermore, in many cases the loop is the last statement of a function, so you can just use a `return` statement to both end the loop and exit the function. (A `return` exits the function that contains it even if execution is in the middle of a conditional or loop.) Using a `return` instead of `break` is more convenient when each function does just one thing: most uses of `break` are intended to move past the loop to execute code that appears later in the function, and if there isn't any code later in the function a `return` statement inside the loop is equivalent to a `break`.

An error that occurs during the execution of a loop's test or one of its statements also terminates the execution of the loop. Altogether, then, there are three ways for a loop's execution to end:

Normally
: The test evaluates to false.

Abnormally
: An error occurs in the evaluation of the test or body of the loop.

Prematurely
: The body of the loop executes a `return` or `break` statement.

When you write a loop, you must make sure that the test expression eventually becomes false or a `break` or `return` is executed. Otherwise, the program will get stuck in what is called an *infinite loop*. We'll see at the end of the chapter how to control what happens when errors occur, rather than allowing them to cause the program to exit abnormally.

Simple Loop Examples

Example 4-1 presents the simplest possible loop, along with a function that reads a line typed by the user, prints it out, and returns it.

Example 4-1. Echo

```
def echo():
    """Echo the user's input until an empty line is entered"""
    while echo1():
        pass

def echo1():
    """Prompt the user for a string, "echo" it, and return it"""
    line = input('Say something: ')
    print('You said', line)
    return line
```

The function `echo1` reads a line, prints it, and returns it. The function `echo` contains the simplest possible `while` statement. It calls a function repeatedly, doing nothing (`pass`),

until the function returns something false. If the user just presses Return, `echo1` will print and return an empty string. Since empty strings are false, when the `while` gets an empty string back from `echo1` it stops. A slight variation, shown in Example 4-2, is to compare the result returned from `echo1` to some specified value that signals the end of the conversation.

Example 4-2. Polite echo

```
def polite_echo():
    """Echo the user's input until it equals 'bye'"""
    while echo1() != 'bye':
        pass
```

Of course, the bodies of loops are rarely so trivial. What allows the loop in these examples to contain nothing but a `pass` is that `echo1` is called both to perform an action and to return `True` or `False`. This example uses such trivial loop bodies only to illustrate the structure of the `while` statement.

Initialization of Loop Values

Example 4-3 shows a more typical loop. It records the user's responses, and when the user types `'bye'` the function returns a record of the input it received. The important thing here is that it's not enough to use `echo1`'s result as a test. The function also needs to add it to a list it is building. That list is returned from the function after the loop exits.

Example 4-3. Recording echo

```
def recording_echo():
    """Echo the user's input until it equals 'bye', then return a list of all the inputs received"""
    lst = []
    entry = echo1()
    while entry != 'bye':
        lst.append(entry)
        entry = echo1()
    return lst
```

In this example, `echo1` is called in two places: once to get the first response and then each time around the loop. Normally it is better not to repeat a piece of code in two places, even if they are so close together. It's easy to forget to change one when you change the other or to make incompatible changes, and changing the same thing in multiple places is tedious and error-prone. Unfortunately, the kind of repetition shown in this example is often difficult to avoid when combining input—whether from the user or from a file—with `while` loops.

TEMPLATE

Function with a Simple Loop

As small as Example 4-3 is, it demonstrates all the usual parts of functions containing a `while` statement.

```
def fn():
    initialize values
    while test values:
        use values
        change values
        # repeat
    return result
```

Example 4-4 shows the same function as Example 4-3, but with comments added to emphasize the way the code uses a simple loop.

Example 4-4. Commented recording echo function

```
def recording_echo():

    # initialize entry and lst
    lst = []

    # get the first input
    entry = echo1()

    # test entry
    while entry != 'bye':

        # use entry
        lst.append(entry)

        # change entry
        entry = echo1()

        # repeat

    # return result
    return lst
```

All parts of this template are optional except for the line beginning with `while`. Typically, one or more of the values assigned in the initialization portion are used in the loop test and changed inside the loop. In `recording_echo` the value of `entry` is initialized, tested, used, and changed; `lst` is initialized, used, and changed, but it is not part of the loop's test.

Looping Forever

Sometimes you just want your code to repeat something until it executes a `return` statement. In that case there's no need to actually test a value. Since `while` statements require a test, we use `True`, which is, of course, always true. This may seem a bit odd, but there are times when something like this is appropriate. It is often called "looping forever." Of course, in reality the program won't run "forever," but it might run forever as far as *it* is concerned—that is, until something external causes it to stop. Such programs are found frequently in operating system and server software.

TEMPLATE

Loop Forever

A conditional's loop expression can be as simple as a single true value, causing it to loop until an external event stops the program.

```
initialize values
while True:
    change values
    if test values:
        return
    use values
    # repeat
return result
```

Example 4-5 shows a rewrite of Example 4-3 using the Loop Forever template. Typical loops usually get the next value at the end of the loop, but in this kind, the next value is obtained at the beginning of the loop.

Example 4-5. Recording echo using "loop forever"

```
def recording_echo_with_conditional():
    """Echo the user's input until it equals 'bye', then return a list of all the inputs received"""
    seq = []
    # no need to initialize a value to be tested since nothing is tested!
    while True:
        entry = echo1()
        if entry == 'bye':
            return seq
        seq.append(entry)
```

Loops over generators are always effectively "forever" in that there's no way to know how many items the generator will produce. The program must call `next` over and over again until the generator is exhausted. We saw in Chapter 3 (in "Generators" on page 78) that the generator argument of `next` can be followed by a value to return when the generator is exhausted. A "forever" loop can be written to use this feature in a function that combines all of the generated amino acid abbreviations into a string. Example 4-6 repeats the definition of the generator function and shows the definition of a new function that uses it.

Example 4-6. Looping over a generator of amino acid symbols

```
def aa_generator(rnaseq):
    """Return a generator object that produces an amino acid by translating
    the next three characters of rnaseq each time next is called on it"""
    return (translate_RNA_codon(rnaseq[n:n+3])
            for n in range(0, len(rnaseq), 3))

def translate(rnaseq):
    """Translate rnaseq into amino acid symbols"""
    gen = aa_generator(rnaseq)
```

```
    seq = ''
    aa = next(gen, None)
    while aa:
        seq += aa
        aa = next(gen, None)
    return seq
```

Loops with Guard Conditions

Loops are often used to search for a value that meets the test condition when there is no guarantee that one does. In such situations it is not enough to just test each value—when there are no more values, the test would be repeated indefinitely. A second conditional expression must be added to detect that no more values remain to be tested.

> **TEMPLATE**
>
> ### Search Loop
>
> In a loop that searches for a value that meets a particular test, the loop condition takes the form:
>
> ```
> not end and not test
> ```
>
> The loop stops when either there are no more values (*end* is true) or a value passes the test. If the function is intended to return a value it must test *end* again after the loop, returning None if it is true and the value that met the test if it is false.
>
> ```
> initialize values
> while not at-end and not at-target:
>     use current values
>     get new values
>     # repeat
> return success-result if test else None
> ```

Loops like these are used when there are two separate reasons for them to end: either there are no more values to use—*at-end*—or some kind of special value has been encountered, detected by *at-target*. If there are no more values to consider, evaluating *at-target* would be meaningless, or, as is often the case, would cause an error. The and operator is used to "protect" the second part of the test so that it is evaluated only if the first is true. This is sometimes called a *guard condition*.

When a loop can end for more than one reason, the statements after the while will need to distinguish the different cases. The simplest and most common case is to return one value if the loop ended because *at-end* became true and a different value if the loop ended because *at-target* became true.

Two-condition loops like this occur frequently in code that reads from streams such as files, terminal input, network connections, and so on. That's because the code cannot know when it has reached the end of the data until it tries to read past it. Before the result of a read can be used, the code must check that something was actually read.

Because `readline` returns `'\n'` when it reads a blank line but returns `''` at the end of a file, it is sufficient to check to see that it returned a nonempty string. Repeated calls to `readline` at the end of the file will continue to return empty strings, so if its return value is not tested the loop will never terminate.

Example 4-7 shows a function that reads the first sequence from a FASTA file. Each time it reads a line it must check first that the line is not empty, indicating that the end of file has been reached, and if not, that the line does not begin with `'>'`, indicating the beginning of the next sequence.

Example 4-7. Checking for a result after every read

```
def read_sequence(filename):
    """Given the name of a FASTA file named filename, read and return
    its first sequence, ignoring the sequence's description"""
    seq = ''
    with open(filename) as file:
        line = file.readline()
        while line and line[0] == '>':
            line = file.readline()
        while line and line[0] != '>':              # must check for end of file
            seq += line
            line = file.readline()
    return seq
```

> Although files can often be treated as collections of lines using comprehensions or `readlines`, in some situations it is more appropriate to loop using `readline`. This is especially true when several related functions all read from the same stream.

The bare outline of code that loops over the lines of a file, doing something to each, is shown in the next template.

TEMPLATE

Looping Over Lines of a File

When several related functions each loop over lines of a file while testing for an end condition that terminates their processing, each must check whether the end of the file has been reached before performing its test. Each function's loop would have the form:

```
line = file.readline()
while line and not test(line):
    do something with line
    line = file.readline()
```

Iterations

Collections contain objects; that's more or less *all* they do. Some built-in functions—`min`, `max`, `any`, and `all`—work for any type of collection. The operators `in` and `not in` accept any type of collection as their second operand. These functions and operators have something very important in common: they are based on doing something with each element of the collection.‡ Since any element of a collection could be its minimum or maximum value, `min` and `max` must consider all the elements. The operators `in` and `not in` and the functions `any` and `all` can stop as soon as they find an element that meets a certain condition, but if none do they too will end up considering every element of the collection.

Doing something to each element of a collection is called *iteration*. We've actually already seen a form of iteration—comprehensions. Comprehensions "do something" to every element of a collection, collecting the results of that "something" into a set, list, or dictionary. Comprehensions with multiple `for` clauses perform nested iterations. Comprehensions with one or more `if` clauses perform conditional iteration: unless an element passes all the `if` tests, the "something" will not be performed for that element and no result will be added.

The code in this book uses comprehensions much more aggressively than many Python programmers do. You should get comfortable using them, because in applicable situations they say just what they mean and say it concisely. Their syntax emphasizes the actions and tests performed on the elements. They produce collection objects, so the result of a comprehension can be used in another expression, function call, `return` statement, etc. Comprehensions help reduce the littering of code with assignment statements and the names they bind.

The question, then, is: what kinds of collection manipulations do *not* fit the mold of Python's comprehensions? From the point of view of Python's language constructs, the answer is that actions performed on each element of a collection sometimes must be expressed using statements, and comprehensions allow only expressions. Comprehensions also can't stop before the end of the collection has been reached, as when a target value has been located. For these and other reasons Python provides the `for` statement to perform general-purpose iteration over collections.

Iteration Statements

Iteration statements all begin with the keyword `for`. This section shows many ways `for` statements can be used and "templates" that summarize the most important.

‡ The `len` function is a bit different: while it could be implemented by counting the elements one at a time, most types implement length more directly.

STATEMENT

Iteration

Python's `for` statement expresses iteration succinctly:

```
for item in collection:
    do something with item
```

You will use `for` statements often, since so much of your programming will use collections. The `for` statement makes its purpose very clear. It is easy to read and write and minimizes the opportunities for making mistakes. Most importantly, it works for collections that aren't sequences, such as sets and dictionaries. As a matter of fact, the `for` statement isn't even restricted to collections: it works with objects of a broader range of types that together are categorized as *iterables*. (For instance, we are treating file objects as collections (streams), but technically they are another kind of iterable.)

> The `continue` and `break` statements introduced in the section on loops work for iterations too.

TEMPLATE

File Iteration

Doing something with each line of a file object is an especially useful application of the `for` statement.

```
with open(filename) as file:
    for line in file:
        do something with line
```

By default, a dictionary iteration uses the dictionary's keys. If you want the iteration to use its values, call the `values` method explicitly. To iterate with both keys and values at the same time, call the `items` method. Typically, when using both keys and values you would unpack the result of `items` and assign a name to each, as shown at the end of the following template.

TEMPLATE

Dictionary Iteration

Iteration can be performed with a dictionary's keys, values, or key/value pairs.

```
for key in dictionary.keys():
    do something with key
```

```
for value in dictionary.values():
    do something with value
```

While an iteration that uses a dictionary's items could begin with `for item in`, it is usually preferable to use tuple unpacking to name the key and value in each item tuple.

```
for key, value in dictionary.items():
    do something with key and value
```

The previous chapter pointed out that if you need a dictionary's keys, values, or items as a list you can call `list` on the result of the corresponding method. This isn't necessary in `for` statements—the results of the dictionary methods can be used directly. `keys`, `values`, and `items` each return an iterable of a different type—`dict_keys`, `dict_values`, and `dict_items`, respectively—but this difference almost never matters, since the results of calls to these methods are most frequently used in `for` statements and as arguments to `list`.

Sometimes it is useful to generate a sequence of integers along with the values over which a `for` statement iterates. The function `enumerate(`*`iterable`*`)` generates tuples of the form `(`*`n, value`*`)`, with *`n`* starting at `0` and incremented with each value taken from the iterable. It is rarely used anywhere but in a `for` statement.

TEMPLATE

Numbering Iterations

Use the function `enumerate` and tuple unpacking to generate numerical keys in parallel with the values in an iterable.

```
for n, value in enumerate(iterable):
    do something with n and value
```

A common use for `enumerate` is to print out the elements of a collection along with a sequence of corresponding numbers. The "do something" line of the template becomes a call to `print` like the following:

```
print(n, value, sep='\t')
```

Kinds of Iterations

Most iterations conform to one of a small number of patterns. Templates for these patterns and examples of their use occupy most of the rest of this chapter. Many of these iteration patterns have correlates that use loops. For instance, Example 4-3 is just like the Collect template described shortly. In fact, anything an iteration can do can be done with loops. In Python programming, however, loops are used primarily to deal with external events and to process files by methods other than reading them line by line.

Iteration should always be preferred over looping. Iterations are a clearer and more concise way to express computations that use the elements of collections, including streams. Writing an iteration is less error-prone than writing an equivalent loop, because there are more details to "get right" in coding a loop than in an iteration.

Do

Often, you just want to do something to every element of a collection. Sometimes that means calling a function and ignoring its results, and sometimes it means using the element in one or more statements (since statements don't have results, there's nothing to ignore).

TEMPLATE

Do

The simplest kind of iteration just does something for each element of a collection.

```
for item in collection:
    do something with item
```

A very useful function to have is one that prints every element of a collection. When something you type into the interpreter returns a long collection, the output is usually difficult to read. Using `pprint.pprint` helps, but for simple situations the solution demonstrated in Example 4-8 suffices. Both `pprint` and this definition can be used in other code too, of course.

Example 4-8. Doing something to (print) each element

```
def print_collection(collection):
    for item in collection:
        print(item)
    print()
```

Actually, even this action could be expressed as a comprehension:

```
[print(item) for item in collection]
```

Since `print` returns `None`, what you'd get with that comprehension is a list containing one `None` value for each item in the collection. It's unlikely that you'd be printing a list large enough for it to matter whether you constructed another one, but in another situation you might call some other no-result function for a very large collection. However, that would be silly and inefficient. What you could do in that case is use a set, instead of list, comprehension:

```
{print(item)for item in collection}
```

That way the result would simply be {None}. Really, though, the use of a comprehension instead of a Do iteration is not a serious suggestion, just an illustration of the close connection between comprehensions and iterations.

We can create a generalized "do" function by passing the "something" as a functional argument, as shown in Example 4-9.

Example 4-9. A generalized do function

```
def do(collection,  fn):
    for item in collection:
        fn(item)
```

The function argument could be a named function. For instance, we could use `do` to redefine `print_collection` from Example 4-8, as shown in Example 4-10.

Example 4-10. Redefining print_collection using a generalized do

```
def print_collection(collection):
    do(collection, print)
```

This passes a named function as the argument to `do`. For more ad hoc uses we could pass a lambda expression, as in Example 4-11.

Example 4-11. Passing a lambda expression to do

```
do(collection, lambda elt: print('\t', elt, sep=''))
```

The way to express a fixed number of repetitions of a block of code is to iterate over a range, as shown in the following template.

TEMPLATE

Repeat

To repeat a block of statements *n* times, iterate over `range(n)`.

```
for count in range(n):
    statements
```

Frequently, `count` would not even be used in the body of the iteration.

Collect

Iterations often collect the results of the "something" that gets done for each element. That means creating a new list, dictionary, or set for the purpose and adding the results to it as they are computed.

TEMPLATE

Collect

A Collect iteration starts with an empty collection and uses a method or operator appropriate to its type to add something to it for each iteration.

```
result = []
for item in collection:
    statements using item
    result.append(expression based on the statements)
return result
```

Most situations in which iteration would be used to collect results would be better expressed as comprehensions. Sometimes, though, it can be tricky to program around the limitation that comprehensions cannot contain statements. In these cases, a Collect iteration may be more straightforward. Perhaps the most common reason to use a Collect iteration in place of a comprehension or loop is when one or more names are assigned and used as part of the computation. Even in those cases, it's usually better to extract that part of the function definition and make it a separate function, after which a call to that function can be used inside a comprehension instead of an iteration.

Example 4-12 shows a rewrite of the functions for reading entries from FASTA files in Chapter 3. In the earlier versions, all the entries were read from the file and then put through a number of transformations. This version, an example of the Collect iteration template, reads each item and performs all the necessary transformations on it before adding it to the collection. For convenience, this example also repeats the most succinct and complete comprehension-based definition.

While more succinct, and therefore usually more appropriate for most situations, the comprehension-based version creates several complete lists as it transforms the items. Thus, with a very large FASTA file the comprehension-based version might take a lot more time or memory to execute. After the comprehension-based version is yet another, this one using a loop instead of an iteration. You can see that it is essentially the same, except that it has extra code to read the lines and check for the end of the file.

Example 4-12. Reading FASTA entries with a Collect iteration

```
def read_FASTA_iteration(filename):
    sequences = []
    descr = None
    with open(filename) as file:
        for line in file:
            if line[0] == '>':
                if descr:                              # have we found one yet?
                    sequences.append((descr, seq))
                descr = line[1:-1].split('|')
                seq = ''                               # start a new sequence
            else:
                seq += line[:-1]
```

```
        sequences.append((descr, seq))              # add the last one found
    return sequences

def read_FASTA(filename):
    with open(filename) as file:
        return [(part[0].split('|'),
                 part[2].replace('\n', ''))
                for part in
                [entry.partition('\n')
                 for entry in file.read().split('>')[1:]]]

def read_FASTA_loop(filename):
    sequences = []
    descr = None
    with open(filename) as file:
        line = file.readline()[:-1]                 # always trim newline
        while line:
            if line[0] == '>':
                if descr:                           # any sequence found yet?
                    sequences.append((descr, seq))
                descr = line[1:].split('|')
                seq = ''                            # start a new sequence
            else:
                seq += line
            line = file.readline()[:-1]
        sequences.append((descr, seq))              # easy to forget!
    return sequences
```

Combine

Sometimes we want to perform an operation on all of the elements of a collection to yield a single value. An important feature of this kind of iteration is that it must begin with an initial value. Python has a built-in `sum` function but no built-in `product`; Example 4-13 defines one.

Example 4-13. A definition of product

```
def product(coll):
    """Return the product of the elements of coll converted to floats, including
    elements that are string representations of numbers; if coll has an element
    that is a string but doesn't represent a number, an error will occur"""
    result = 1.0                                    # initialize
    for elt in coll:
        result *= float(elt)                        # combine element with
    return result                                   # accumulated result
```

As simple as this definition is, there is no reasonable way to define it just using a comprehension. A comprehension always creates a collection—a set, list, or dictionary—and what is needed here is a single value. This is called a Combine (or, more technically, a "Reduce"§) because it starts with a collection and ends up with a single value.

§ The term "reduce" comes from the mathematical idea that a one-dimensional collection is *reduced* to a "zero"-dimensional "scalar" value.

TEMPLATE

Combine

The general outline for a Combine iteration is as follows, with · representing a binary operation and *fn* a two-parameter function. The *identity value* is the value *i* for which *i* · *v* == *v* or *fn*(*i*,*v*) == *v*. Typical examples would be 0 for addition and 1 for multiplication. When collections are being combined, the identity element is an empty collection.

Only one of the three forms shown inside the for statement would be used; they are shown here together for convenience.

```
result = identity-value
for item in collection:
    result = result · item           # One of these
    result ·= item                   # three forms
    result =  fn(result, item)       # is used.
return result
```

For another example, let's find the longest sequence in a FASTA file. We'll assume we have a function called read_FASTA, like one of the implementations shown in Chapter 3. Example 4-13 used a binary operation to combine each element with the previous result. Example 4-14 uses a two-valued function instead, but the idea is the same. The inclusion of an assignment statement inside the loop is an indication that the code is doing something that cannot be done with a comprehension.

Example 4-14. Combine: identifying the longest FASTA sequence

```
def longest_sequence(filename):
    longest_seq = ''
    for info, seq in read_FASTA(filename):
        longest_seq = max(longest_seq, seq, key=len)
    return longest_seq
```

A special highly reduced form of Combine is Count, where all the iteration does is count the number of elements. It would be used to count the elements in an iterable that doesn't support length. This template applies particularly to generators: for a generator that produces a large number of items, this is far more efficient than converting it to a list and then getting the length of the list.

TEMPLATE

Count

A Count iteration "combines" the value 1 for each element of the iteration.

```
count = 0
for item in iterable:
    count += 1
return count
```

One of the most important and frequently occurring kinds of actions on iterables that cannot be expressed as a comprehension is one in which the result of doing something to each element is itself a collection (a list, usually), and the final result is a combination of those results. An ordinary Combine operation "reduces" a collection to a value; a Collection Combine reduces a *collection of collections* to a *single collection*. (In the template presented here the reduction is done step by step, but it could also be done by first assembling the entire collection of collections and then reducing them to a single collection.)

TEMPLATE

Collection Combine

In this variation on Combine, an action is performed on each element of a collection that produces a collection as a result, but instead of returning a collection of the results, the iteration combines the results into (reduces the results to) a single collection.

```
result = []
for item in collection:
    result += fn(item)
    # merge result with previous results
return result
```

Example 4-15 shows an example in which "GenInfo" IDs are extracted from each of several files, and a single list of all the IDs found is returned.

Example 4-15. Collection Combine: sequence IDs from multiple files

```
def extract_gi_id(description):
    """Given a FASTA file description line, return its GenInfo ID if it has one"""
        if line[0] != '>':
        return None
    fields = description[1:].split('|')
    if 'gi' not in fields:
        return None
    return fields[1 + fields.index('gi')]

def get_gi_ids(filename):
    """Return a list of the GenInfo IDs of all sequences found in the file named filename"""
    with open(filename) as file:
        return [extract_gi_id(line) for line in file if line[0] == '>']

def get_gi_ids_from_files(filenames):
    """Return a list of the GenInfo IDs of all sequences found in the
    files whose names are contained in the collection filenames"""
    idlst = []
    for filename in filenames:
        idlst += get_gi_ids(filename)
    return idlst
```

Search

Another common use of iterations is to search for an element that passes some kind of test. This is not the same as a combine iteration—the result of a combination is a property of all the elements of the collection, whereas a search iteration is much like a search loop. Searching takes many forms, not all of them iterations, but the one thing you'll just about always see is a `return` statement that exits the function as soon as a matching element has been found. If the end of the function is reached without finding a matching element the function can end without explicitly returning a value, since it returns `None` by default.

> **TEMPLATE**
>
> **Search**
>
> Search is a simple variation on Do:
>
> ```
> for item in collection:
>     if test item:
>         return item
> ```

Suppose we have an enormous FASTA file and we need to extract from it a sequence with a specific GenBank ID. We don't want to read every sequence from the file, because that could take much more time and memory than necessary. Instead, we want to read one entry at a time until we locate the target. This is a typical search. It's also something that comprehensions can't do: since they can't incorporate statements, there's no straightforward way for them to stop the iteration early.

As usual, we'll build this out of several small functions. We'll define four functions. The first is the "top-level" function; it calls the second, and the second calls the third and fourth. Here's an outline showing the functions called by the top-level function:

```
search_FASTA_file_by_gi_id(id, filename)
    FASTA_search_by_gi_id(id, fil)
        extract_gi_id(line)
        read_FASTA_sequence(fil)
```

This opens the file and calls `FASTA_search_by_gi_id` to do the real work. That function searches through the lines of the file looking for those beginning with a `'>'`. Each time it finds one it calls `get_gi_id` to get the GenInfo ID from the line, if there is one. Then it compares the extracted ID to the one it is looking for. If there's a match, it calls `read_FASTA_sequence` and returns. If not, it continues looking for the next FASTA description line. In turn, `read_FASTA_sequence` reads and joins lines until it runs across a description line, at which point it returns its result. Example 4-16 shows the definition of the top-level function.

"Top-level" functions should almost always be very simple. They are entry points into the capabilities the other function definitions provide. Essentially, what they do is

prepare the information received through their parameters for handling by the functions that do the actual work.

Example 4-16. Extracting a sequence by ID from a large FASTA file

```
def search_FASTA_file_by_gi_id(id, filename):
    """Return the sequence with the GenInfo ID ID from the FASTA file
    named filename, reading one entry at a time until it is found"""
    id = str(id)                                  # user might call with a number
    with open(filename) as file:
        return FASTA_search_by_gi_id(id, file)
```

Each of the other functions can be implemented in two ways. Both `FASTA_search_by_gi_id` and `read_FASTA_sequence` can be implemented using a loop or iteration. The simple function `get_gi_id` can be implemented with a conditional expression or a conditional statement. Table 4-1 shows both implementations for `FASTA_search_by_gi_id`.

Table 4-1. Two definitions of FASTA_search_by_gi_id

<table>
<tr><th>Iteration</th><th>Loop</th></tr>
<tr><td colspan="2"><code>def FASTA_search_by_gi_id(id, file):</code></td></tr>
<tr><td><pre>
for line in file:
    if (line[0] == '>' and
        str(id) ==   get_gi_id(line)):
            return \
               read_FASTA_sequence(file)
</pre></td><td><pre>
line = file.readline()
while (line and
        not (line[0] == '>' and
             (str(id) ==
                get_gi_id(line)))):
             line = file.readline()
return (line and
        read_FASTA_sequence(fil)
</pre></td></tr>
</table>

The iterative implementation of `FASTA_search_by_gi_id` treats the file as a collection of lines. It tests each line to see if it is the one that contains the ID that is its target. When it finds the line it's seeking, it does something slightly different than what the template suggests: instead of returning the line—the item found—it goes ahead and reads the sequence that follows it.

> The templates in this book are not meant to restrict your code to specific forms: they are frameworks for you to build on, and you can vary the details as appropriate.

The next function—`read_FASTA_sequence`—shows another variation of the search template. It too iterates over lines in the file—though not all of them, since it is called after `FASTA_search_by_gi_id` has already read many lines. Another way it varies from the template is that it accumulates a string while looking for a line that begins with a `'>'`. When it finds one, it returns the accumulated string. Its definition is shown in Table 4-2, and the definition of `get_gi_id` is shown in Table 4-3.

Table 4-2. Two definitions of read_FASTA_sequence

Iteration	Loop
def read_FASTA_sequence(file): seq = ''	
for line in file: **if** not line or line[0] == '>': **return** seq seq += line[:-1]	line = file.readline() **while** line and line[0] != '>': seq += line[:-1] line = file.readline() **return** seq

Table 4-3. Two definitions of get_gi_id

Conditional statement	Conditional expression
def get_gi_id(description): fields = description[1:].split('|')	
if fields and 'gi' in fields: **return** fields[(1 + fields.index('gi')]	**return** (fields and 'gi' in fields and fields[1+fields.index('gi')])

A special case of search iteration is where the result returned is interpreted as a Boolean rather than the item found. Some search iterations return `False` when a match is found, while others return `True`. The exact form of the function definition depends on which of those cases it implements. If finding a match to the search criteria means the function should return `False`, then the last statement of the function would have to return `True` to show that all the items had been processed without finding a match. On the other hand, if the function is meant to return `True` when it finds a match it is usually not necessary to have a `return` at the end, since `None` will be returned by default, and `None` is interpreted as false in logical expressions. (Occasionally, however, you really need the function to return a Boolean value, in which case you would end the function by returning `False`.) Here are two functions that demonstrate the difference:

```
def rna_sequence_is_valid(seq):
    for base in seq:
        if base not in 'UCAGucag':
            return False
    return True

def dna_sequence_contains_N(seq):
    for base in seq:
    if base == 'N':
        return True
```

Filter

Filtering is similar to searching, but instead of returning a result the first time a match is found, it does something with each element for which the match was successful. Filtering doesn't stand on its own—it's a modification to one of the other kinds of iterations. This section presents templates for some filter iterations. Each just adds a

conditional to one of the other templates. The condition is shown simply as ***test*** **item**, but in practice that test could be complex. There might even be a few initialization statements before the conditional.

TEMPLATE

Filtered Do

A Filtered Do performs an action for each item that meets a specified condition.

```
for item in collection:
    if test item:
        statements using item
```

An obvious example of a Filtered Do is printing the header lines from a FASTA file. Example 4-17 shows how this would be implemented.

Example 4-17. Printing the header lines from a FASTA file

```
def print_FASTA_headers(filename):     with open(filename) as file:
        for line in file:
            if line[0] == '>':
                print(line[1:-1])
```

As with Collect iterations in general, simple situations can be handled with comprehensions, while iterations can handle the more complex situations in which statements are all but unavoidable. For example, extracting and manipulating items from a file can often be handled by comprehensions, but if the number of items is large, each manipulation will create an unnecessarily large collection. Rather than collecting all the items and performing a sequence of operations on that collection, we can turn this inside out, performing the operations on one item and collecting only the result.

TEMPLATE

Filtered Collect

Here, the values for which ***test*** is true are collected one at a time.

```
result = []
for item in collection:
    if test item:
        statements using item
        result.append(expression based on the statements)
return result
```

In many cases, once a line passes the test the function should not return immediately. Instead, it should continue to read lines, concatenating or collecting them, until the next time the test is true. An example would be with FASTA-formatted files, where a function might look for all sequence descriptions that contain a certain string, then

read all the lines of the sequences that follow them. What's tricky about this is that the test applies only to the lines beginning with `'>'`. The lines of a sequence do not provide any information to indicate whether they should be included or not.

Really what we have here are two tests: there's a preliminary test that determines whether the primary test should be performed. Neither applies to the lines that follow a description line in the FASTA file, though. To solve this problem, we add a flag to govern the iteration and set it by performing the primary test whenever the preliminary test is true. Example 4-18 shows a function that returns the sequence strings for all sequences whose descriptions contain the argument `string`.

Example 4-18. Extracting sequences with matching descriptions

```
def extract_matching_sequences(filename, string):
    """From a FASTA file named filename, extract all sequences whose descriptions contain string"""
    sequences = []
    seq = ''
    with open(filename) as file:
        for line in file:
            if line[0] == '>':
                if seq:                                 # not first time through
                    sequences.append(seq)
                seq = ''          # next sequence detected
                includeflag = string in line            # flag for later iterations
            else:
                if includeflag:
                    seq += line[:-1]
        if seq:                                         # last sequence in file is included
            sequences.append(seq)
    return sequences
```

The generalization of this code is shown in the following template.

TEMPLATE

Filtered Collect of Groups of Lines

The details of code implementing this template vary more than the template implies. Its overall job is to look for a line that meets both a preliminary test and a primary test. It collects or concatenates lines when the primary test is true until the next line for which the preliminary test is true. This requires a flag that keeps track of the result of the primary test while subsequent lines are read.

```
lines = []
with open(inputfilename) as file:
    for line in file:
        if preliminary-test:
            flag = primary-test(line)
            lines.append(line)                  # or concatenate, etc.
return lines
```

A Filtered Combine is just like a regular Combine, except only elements that pass the test are used in the combining expression.

TEMPLATE

Filtered Combine

As with a regular Combine, only one of the three forms shown inside the condition is used for a given piece of code implementing this template.

```
result = identity-value
for item in collection:
    if test item:
        result = result · item                    # One of these
        result ·= item                            # three forms
        result =  fn(result, item)                # is used.
return result
```

Example 4-13 showed a definition for `product`. Suppose the collection passed to `product` contained nonnumerical elements. You might want the `product` function to skip nonnumerical values instead of converting string representations of numbers to numbers.[‖]

All that's needed to skip nonnumerical values is a test that checks whether the element is an integer or float and ignores it if it is not. The function `isinstance` was described briefly in Chapter 1; we'll use that here to check for numbers. Example 4-19 shows this new definition for `product`.

Example 4-19. Filtered Combine: another definition of product

```
def is_number(value):
    """Return True if value is an int or a float"""
    return isinstance(elt, int) or isinstance(elt, float)

def product(coll):
    """Return the product of the numeric elements of coll"""
    result = 1.0                          # initialize
    for elt in coll:
        if is_number(elt):
            result = result * float(elt)  # combine element with accumulated result
    return result
```

What we've done here is replace the template's *`test`* with a call to `is_number` to perform the test. Suppose we needed different tests at different times while computing the product—we might want to ignore zeros or negative numbers, or we might want to start at a different initial value (e.g., `1` if computing the product of only integers). We might even have different actions to perform each time around the iteration. We can

‖ Spreadsheet applications, for example, typically skip nonnumbers when performing numeric operations like "sum" on a row or column, rather than producing an error.

implement many of these templates as function definitions whose details are specified by parameters. Example 4-20 shows a completely general `combine` function.

Example 4-20. Generalized combine function

```
def combine(coll, initval, action, filter=None):
    """Starting at initval, perform action on each element of coll, finally returning the result. If
    filter is not None, only include elements for which filter(element) is true. action is a function
    of two arguments--the interim result and the element--which returns a new interim result."""
    result = initval
    for elt in coll:
        if not filter or filter(elt):
           result = action(result, elt)
    return result
```

To add all the integers in a collection, we just have to call `combine` with the right arguments:

```
combine(coll
        0,
        lambda result, elt: result + elt,
        lambda elt: isinstance(elt, int)
        )
```

TEMPLATE

Filtered Count

An important specific variation of Filtered Combine is Filtered Count. This is useful for collections even if they support the `len` function, because `len` cannot include certain items.

```
count = 0
for item in iterable:
    if test item:
        count += 1
return count
```

Nested iterations

One iteration often uses another. Example 4-21 shows a simple case—listing all the sequence IDs in files whose names are in a collection.

Example 4-21. A nested iteration

```
def list_sequences_in_files(filelist):
    """For each file whose name is contained in filelist,
    list the description of each sequence it contains"""
    for filename in filelist:
        print(filename)
        with open(filename) as file:
            for line in file:
                if line[0] == '>':
                    print('\t', line[1:-1])
```

Nesting is not a question of physical containment of one piece of code inside another. Following the earlier recommendation to write short, single-purpose functions, Example 4-22 divides the previous function, placing one iteration in each. This is still a nested iteration, because the first function calls the second each time around the `for`, and the second has its own `for` statement.

Example 4-22. A two-function nested iteration

```
def list_sequences_in_files(filelist):
    """For each file whose name is contained in filelist,
    list the description of each sequence it contains"""
    for filename in filelist:
        print(filename)
        with open(filename) as file:
            list_sequences_in_file(file)

def list_sequences_in_file(file)
    for line in file:
        if line[0] == '>':
            print('\t', line[1:-1])
```

These examples do present nested iterations, but they don't show what's special about this kind of code. Many functions that iterate call other functions that also iterate. They in turn might call still other functions that iterate. Nested iterations are more significant when their "do something" parts involve doing something with a value from the outer iteration and a value from the inner iteration together.

TEMPLATE

Nested Iteration

The general form of a nested iteration is as follows, keeping in mind that the inner iteration might actually be defined as a separate function.

```
for outer in outer_collection:
    for inner in inner_collection:
        do something with inner and outer
```

Perhaps a batch of samples is to be submitted for sequencing with each of a set of primers:

```
for seq in sequences:
    for primer in primers:
        submit(seq, primer)
```

This submits a sequence and a primer for every combination of a sequence from `sequences` and a primer from `primers`. In this case it doesn't matter which iteration is the outer and which is the inner, although if they were switched the sequence/primer pairs would be submitted in a different order.

Three-level iterations are occasionally useful—especially in bioinformatics programming, because codons consist of three bases. Example 4-23 shows a concise three-level iteration that prints out a simple form of the DNA codon table.

Example 4-23. Printing the codon table

```
def print_codon_table():
    """Print the DNA codon table in a nice, but simple, arrangement"""
    for base1 in DNA_bases:                       # horizontal section (or "group")
        for base3 in DNA_bases:                   # line (or "row")
            for base2 in DNA_bases:               # vertical section (or "column")
                # the base2 loop is inside the base3 loop!
                print(base1+base2+base3,
                      translate_DNA_codon(base1+base2+base3),
                      end='  ')
            print()
        print()
>>> print_codon_table()
TTT Phe  TCT Ser  TAT Tyr  TGT Cys
TTC Phe  TCC Ser  TAC Tyr  TGC Cys
TTA Leu  TCA Ser  TAA ---  TGA ---
TTG Leu  TCG Ser  TAG ---  TGG Trp

CTT Leu  CCT Pro  CAT His  CGT Arg
CTC Leu  CCC Pro  CAC His  CGC Arg
CTA Leu  CCA Pro  CAA Gln  CGA Arg
CTG Leu  CCG Pro  CAG Gln  CGG Arg

ATT Ile  ACT Thr  AAT Asn  AGT Ser
ATC Ile  ACC Thr  AAC Asn  AGC Ser
ATA Ile  ACA Thr  AAA Lys  AGA Arg
ATG Met  ACG Thr  AAG Lys  AGG Arg

GTT Val  GCT Ala  GAT Asp  GGT Gly
GTC Val  GCC Ala  GAC Asp  GGC Gly
GTA Val  GCA Ala  GAA Glu  GGA Gly
GTG Val  GCG Ala  GAG Glu  GGG Gly
```

Recursive iterations

Trees are an important class of data structure in computation: they provide the generality needed to represent branching information. Taxonomies and filesystems are good examples. A filesystem starts at the top-level directory of, say, a hard drive. That directory contains files and other directories, and those directories in turn contain files and other directories. The whole structure consists of just directories and files.

A data structure that can contain other instances of itself is said to be *recursive*. The study of recursive data structures and algorithms to process them is a major subject in computer science. Trees are the basis of some important algorithms in bioinformatics too, especially in the areas of searching and indexing.

While we won't be considering such algorithms in this book, it is important to know some rudimentary techniques for tree representation and iteration. A simple `while` or

`for` statement can't by itself follow all the branches of a tree. When it follows one branch, it may encounter further branches, and at each juncture it can follow only one at a time. It can only move on to the next branch after it's fully explored everything on the first one. In the meantime, it needs someplace to record a collection of the remaining branches to be processed.

Each branch is just another tree. A function that processes a tree can call *itself* to process each of the tree's branches. What stops this from continuing forever is that eventually subtrees are reached that have no branches; these are called *leaves*. A function that calls itself—or calls another function that eventually calls it—is called a *recursive function*.

Discussions of recursion are part of many programming texts and courses. It often appears mysterious until the idea becomes familiar, which can take some time and practice. One of the advantages of recursive functions is that they can express computations more concisely than other approaches, even when recursion isn't actually necessary. Sometimes the code is so simple you can hardly figure out how it does its magic!

First, we'll look at an example of one of the ways trees are used in bioinformatics. Some very powerful algorithms used in indexing and searching genomic sequences rely on what are called *suffix trees*. These are tree structures constructed so that every path from the root to a leaf produces a subsequence that is not the prefix of any other subsequence similarly obtained. The entire string from which the tree was constructed can be recovered by traversing all paths to leaf nodes, concatenating the strings encountered along the way, and collecting the strings obtained from each path. The longest string in the resulting collection is the original string. Figure 4-2 shows an example.

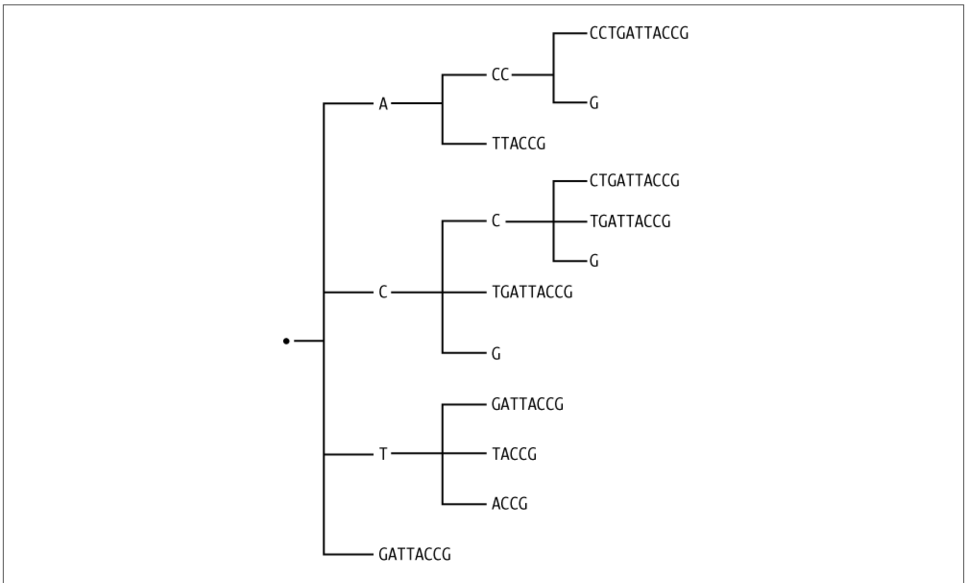


Figure 4-2. Diagram of a suffix tree

Algorithms have been developed for constructing and navigating such trees that do their work in an amount of time that is directly proportional to the length of the sequence. Normally algorithms dealing with tree-structured data require time proportional to N^2 or at best $N \log N$, where N is the length of the sequence. As N gets as large as is often required for genomic sequence searches, those quantities grow impossibly large. From this point of view the performance of suffix tree algorithms borders on the miraculous.

Our example will represent suffix trees as lists of lists of lists of... lists. The first element of each list will always be a string, and each of the rest of the elements is another list. The top level of the tree starts with an empty string. Example 4-24 shows an example hand-formatted to reflect the nested relationships.

Example 4-24. Representing a tree as a list of lists

```
['',
    ['A',
        ['CC',
            ['CCTGATTACCG'],
            ['G']
        ],
        ['TTACCG']
    ],
    ['C',
        ['C',
            ['CTGATTACCG'],
            ['TGATTACCG'],
            ['G']
        ],
        ['TGATTACCG'],
        ['G']
    ],
    ['T',
        ['GATTACCG'],
        ['TACCG'],
        ['ACCG']
    ],
    ['GATTACCG']
]
```

Let's assign `tree1` to this list and see what Python does with it. Example 4-25 shows an ordinary interpreter printout of the nested lists.

Example 4-25. Interpreter printing of nested lists representing a tree

```
['', ['A', ['CC', ['CCTGATTACCG'], ['G']], ['TTACCG']], ['C', ['C', ['CTGATTACCG'],
 ['TGATTACCG'], ['G']], ['TGATTACCG'], ['G']], ['T', ['GATTACCG'], ['TACCG'], ['ACCG']],
 ['GATTACCG']]
```

That output was one line, wrapped. Not very helpful. How much of an improvement does `pprint.pprint` offer?

```
>>> pprint.pprint(tree1)
['',
 ['A', ['CC', ['CCTGATTACCG'], ['G']], ['TTACCG']],
 ['C', ['C', ['CTGATTACCG'], ['TGATTACCG'], ['G']], ['TGATTACCG'], ['G']],
 ['T', ['GATTACCG'], ['TACCG'], ['ACCG']],
 ['GATTACCG']]
```

This is a little better, since we can at least see the top-level structure. But what we want is output that approximates the tree shown in Figure 4-2. (We won't go so far as to print symbols for lines and corners—we're just looking to reflect the overall shape of the tree represented by the data structure.) Here's the template for a recursive function to process a tree represented as just described here. (The information the tree contains could be anything, not just strings: whatever value is placed in the first position of the list representing a subtree is the value of that subtree's root node.)

TEMPLATE

Recursive Tree Iteration

This little bit of code will process all the nodes of a tree represented as a list containing a value followed by a number of subtrees. A tree with no subtrees is a list containing just a value.

```
def treewalk(tree, level=0):
    do something with tree[0] and level
    for subtree in tree[1:]:
        treewalk(node, level+1)
```

Do you find it difficult to believe that so simple a template can process a tree? Example 4-26 shows how it would be used to print our tree.

Example 4-26. Printing a tree

```
def treeprint(tree, level=0):
    print(' ' * 4 * level, tree[0], sep='')
    for node in tree[1:]:
        treeprint(node, level+1)
```

This produces the following output for the example tree. It's not as nice as the diagram; not only are there no lines, but the root of each subtree is on a line before its subtrees, rather than centered among them. Still, it's not bad for four lines of code!

```
    A
        CC
            CCTGATTACCG
            G
        TTACCG
    C
        C
            CTGATTACCG
            TGATTACCG
            G
```

```
        TGATTACCG
        G
T
        GATTACCG
        TACCG
        ACCG
GATTACCG
```

Figures 4-3, 4-4, and 4-5 illustrate the process that ensues as the function in Example 4-26 does its work with the list representing the subtree rooted at `A`.

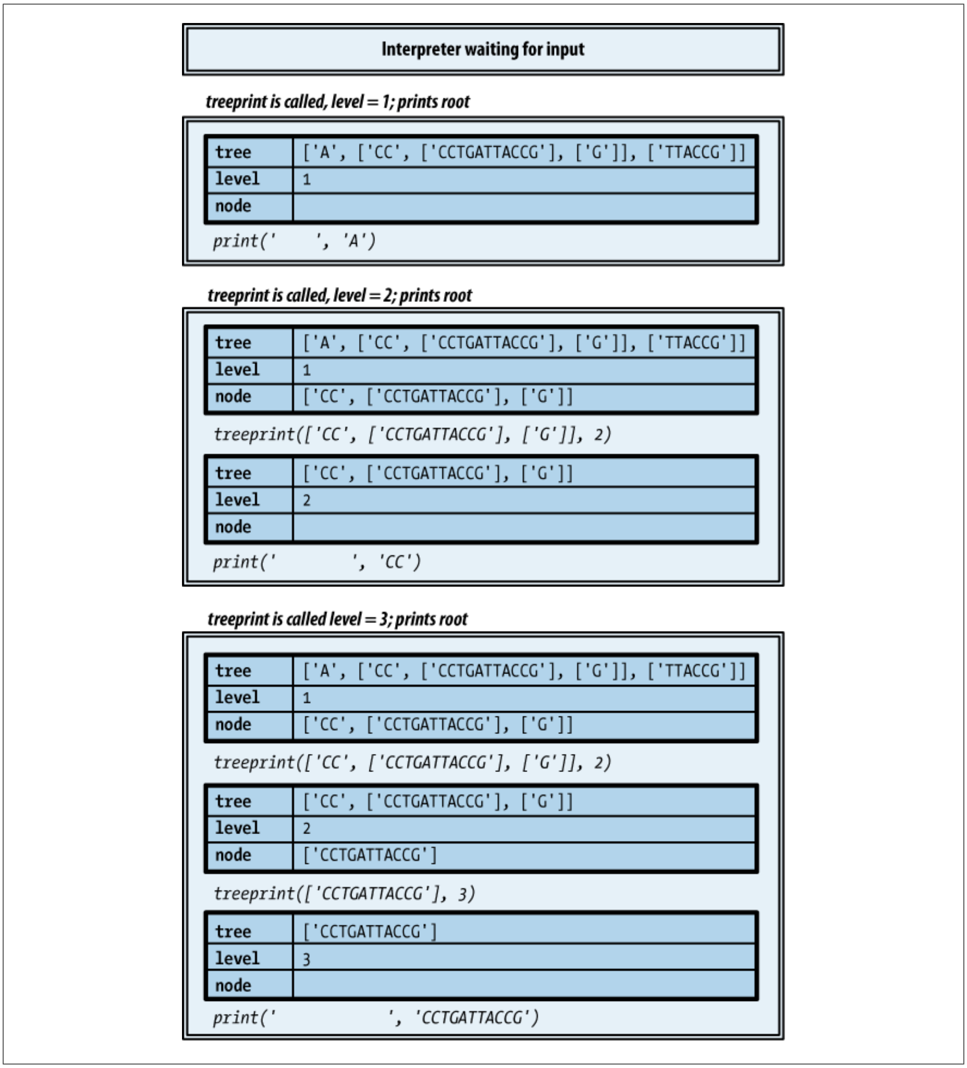


Figure 4-3. Implementation of recursion, part 1

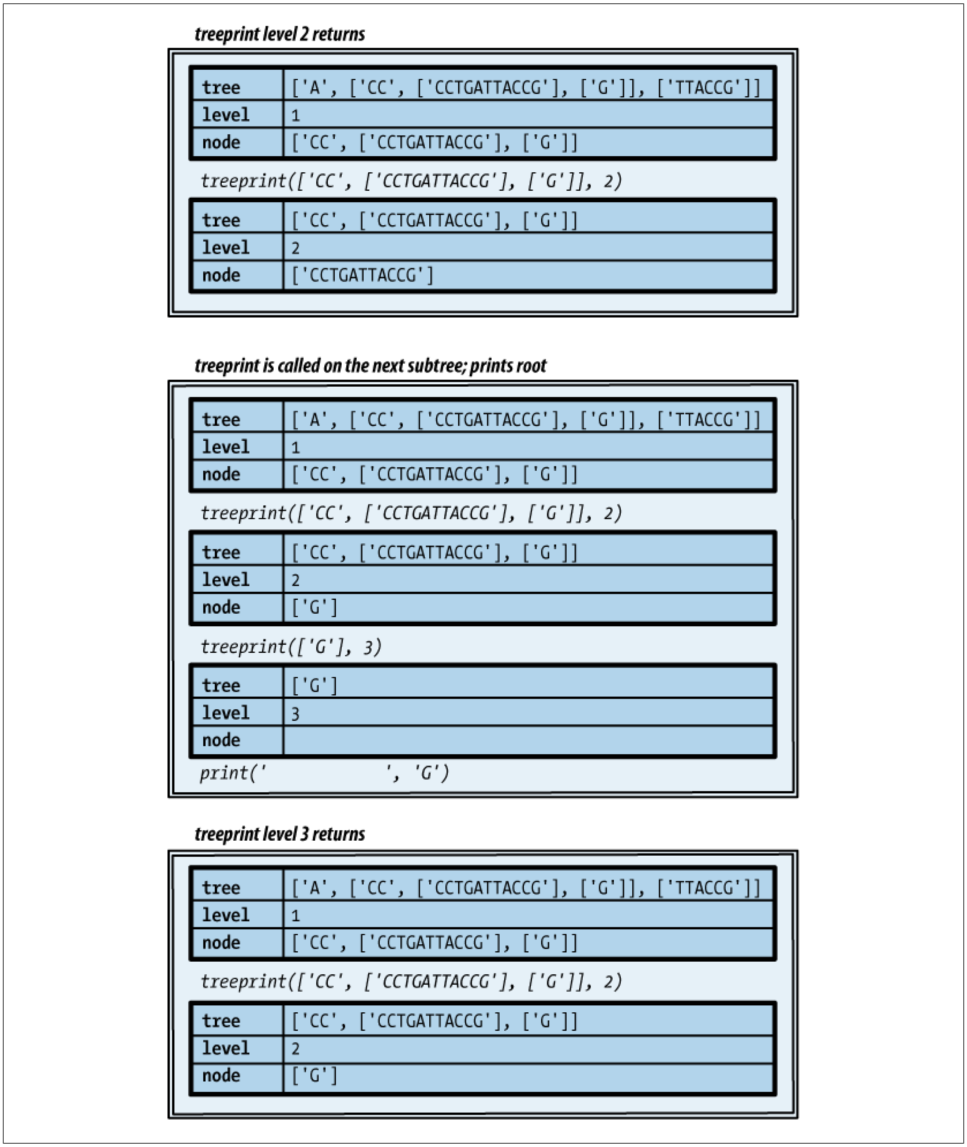


Figure 4-4. Implementation of recursion, part 2

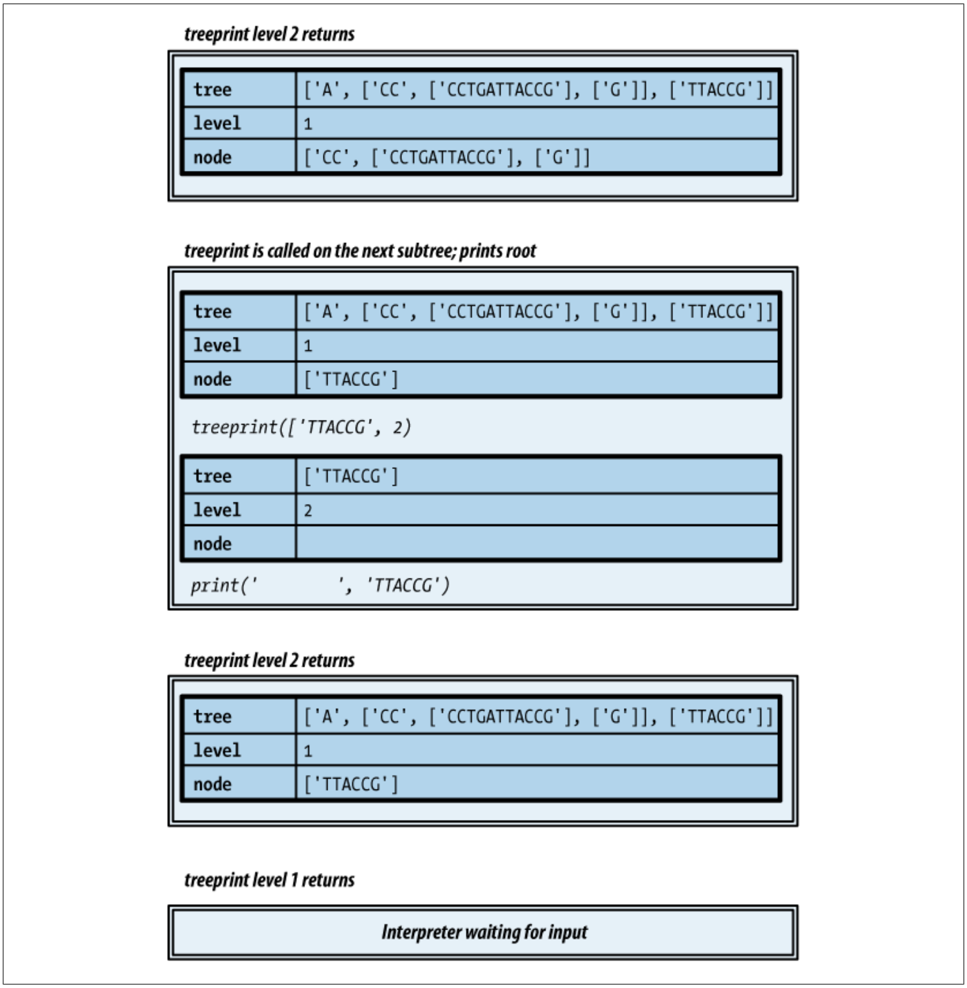


Figure 4-5. Implementation of recursion, part 3

Exception Handlers

Let's return to Example 4-15, from our discussion of collection iteration. We'll add a top-level function to drive the others and put all of the functions in one Python file called *get_gi_ids.py*. The contents of the file are shown in Example 4-27.

Example 4-27. Collecting GenInfo IDs of the sequences in FASTA files

```
def extract_gi_id(description):
    """Given a FASTA file description line, return its GenInfo ID if it has one"""
        if line[0] != '>':
        return None
```

```
    fields = description[1:].split('|')
    if 'gi' not in fields:
        return None
    return fields[1 + fields.index('gi')]

def get_gi_ids(filename):
    """Return a list of GenInfo IDs from the sequences in the FASTA file named filename"""
    with open(filename) as file:
        return [extract_gi_id(line) for line in file if line[0] == '>']

def get_gi_ids_from_files(filenames):
    """Return a list of GenInfo IDs from the sequences in the
    FASTA files whose names are in the collection filenames"""
    idlst = []
    for filename in filenames:
        idlst += get_gi_ids(filename)
    return idlst

def get_gi_ids_from_user_files():
    response = input("Enter FASTA file names, separated by spaces: ")
    lst = get_gi_ids_from_files(response.split())    # assuming no spaces in file names
    lst.sort()
    print(lst)

get_gi_ids_from_user_files()
```

We run the program from the command line, enter a few filenames, and get the results shown in Example 4-28.

Example 4-28. Traceback from attempting to open a nonexistent file

```
% python get_gi_ids.py
Enter a list of FASTA filenames:
aa1.fasta aa2.fasta aa3.fasta

Traceback (most recent call last):
  File "get_gi_ids.py", line 27, in <module>
    get_gi_ids_from_user_files
  File "get_gi_ids.py", line 23, in get_gi_ids_from_user_files
    lst = get_gi_ids_from_files(files))
  File "get_gi_ids.py", line 18, in get_gi_ids_from_files
    idlst += get_gi_ids(filename)
  File "get_gi_ids.py", line 10, in get_gi_ids
    with open(filename) as file:
  File "/usr/local/lib/python3.1/io.py", line 278, in __new__
    return open(*args, **kwargs)
  File "/usr/local/lib/python3.1/io.py", line 222, in open
    closefd)
  File "/usr/local/lib/python3.1/io.py", line 619, in __init__
    _fileio._FileIO.__init__(self, name, mode, closefd)
IOError: [Errno 2] No such file or directory: 'aa2.fasta'
```

Python Errors

If you've executed any Python code you have written, you have probably already seen output like that in the previous example splattered across your interpreter or shell window. Now it's time for a serious look at what this output signifies. It's important to understand more than just the message on the final line and perhaps a recognizable filename and line number or two.

Tracebacks

As its first line indicates, the preceding output shows details of pending functions. This display of information is called a *traceback*. There are two lines for each entry. The first shows the name of the function that was called, the path to the file in which it was defined, and the line number where its definition begins, though not in that order. (As in this case, you will often see `<module>` given as the module name on the first line; this indicates that the function was called from the top level of the file being run by Python or directly from the interpreter.) The second line of each entry shows the text of the line identified by the filename and line number of the first line, to save you the trouble of going to the file to read it.

> Some of this will make sense to you now. Some of it won't until you have more Python knowledge and experience. As the calls descend deeper into Python's implementation, some technical details are revealed that we haven't yet explored. It's important that you resist the temptation to dismiss tracebacks as hopelessly complicated and useless. Even if you don't understand all the details, tracebacks tell you very clearly what has happened, where, and, to some extent, why.

The problem causing the traceback in this example is clear enough: the user included the file *aa2.fasta* in the list of files to be processed, but when `get_gi_id` went to open that file it couldn't find it. As a result, Python reported an `IOError` and stopped executing. It didn't even print the IDs that it had already found—it just stopped.

Runtime errors

Is this what you want your program to do? You can't prevent the user from typing the name of a nonexistent file. While you could check that each file exists before trying to open it (using methods from the `os` module that we'll be looking in Chapter 6), this is only one of the many things that could go wrong during the execution of your program. Maybe the file exists but you don't have read privileges for it, or it exists but is empty, and you didn't write your code to correctly handle that case. Maybe the program encounters an empty line at the end of the file and tries to extract pieces from it. Maybe the program tries to compare incompatible values in an expression such as `4 < '5'`.

By now you've probably encountered `ValueError`, `TypeError`, `IndexError`, `IOError`, and perhaps a few others. Each of these errors is actually a type. Table 4-4 shows examples of common errors, the type of error instance that gets created when they occur, and examples of the messages that get printed.

Table 4-4. Common runtime errors

Example	Error class	Message
`foobah`	`NameError`	`name 'foobah' is not defined`
`3 < '4'`	`TypeError`	`unorderable types: int() < str()`
`['a', 'b'] + None`	`TypeError`	`can only concatenate list (not "NoneType") to list`
`range()`	`TypeError`	`range expected 1 arguments, got 0`
`1/0`	`ZeroDivisionError`	`int division or modulo by zero`
`int('21', 2)`[a]	`ValueError`	`invalid literal for int() with base 2: '21'`
`[1,2,3].index(4)`[b]	`ValueError`	`list.index(x): x not in list`
`''[1]`	`IndexError`	`string index out of range`
`{'a': 1}['b']`	`KeyError`	`'b'`
`range(4).index(5)`	`AttributeError`	`'range' object has no attribute 'index'`
`open('')`	`IOError`	`No such file or directory: ''`
`input('?')^D`[c]	`EOFError`	
`assert False`	`AssertionError`	
`assert 0, 'not 1'`	`AssertionError`	`not 1`
`^C^C`[d]	`KeyboardInterrupt`	

[a] The second argument of `int` is a base to use when reading the string that is its first argument; in this example, since base 2 was specified, only `'0'` and `'1'` are valid in the first argument.

[b] `index` is like `find`, but instead of returning `-1` when its argument is not in the list, it raises a `ValueError`.

[c] Typing Ctrl-D on an empty line (Ctrl-Z on Windows) ends input. Remember, though, that `file.read` and `file.readline` return empty strings if they are at the end of a file.

[d] Pressing Ctrl-C twice stops whatever Python is doing and returns to the interpreter.

Even if `get_gi_ids` was written to detect nonexistent files before trying to open them, what should it do if it detects one? Should it just return `None`? Should it print its own error message before returning `None`? If it returns `None`, how can the function that called it know whether that was because the file didn't exist, couldn't be read, wasn't a FASTA-formatted file, or just didn't have any sequences with IDs? If each function has to report to its caller all the different problems it might have encountered, each caller will have to execute a series of conditionals checking each of those conditions before continuing with its own executions.

To manage this problem, languages provide *exception handling* mechanisms. These make it possible to ignore exceptions when writing most function definitions, while specifically designating, in relatively few places, what should happen when exceptions

do occur. The term "exception" is used instead of "error" because if the program is prepared to handle a situation, it isn't really an error when it arises. It becomes an error—an *unhandled exception*—if the program does not detect the situation. In that case, execution stops and Python prints a traceback with a message identifying the type of error and details about the problem encountered.

Exception Handling Statements

Python's exception handling mechanism is implemented through the `try` statement. This looks and works much like a conditional, except that the conditions are not tests you write, but rather names of error classes.

STATEMENT

Exception Handling

The basic form of a statement that handles an exception is:

```
try:
    try-statements
except ErrorClass:
    except-statements
```

The error class is one of the error names you'll see printed out on the last line of a traceback: `IOError`, `ValueError`, and so on. When a `try` statement begins, it starts executing the statements in the ***`try-statements`*** block. If they complete without any errors, the rest of the `try` statement is skipped and execution continues at the next statement.

However, if an error of the type identified in the `except` clause occurs during the execution of the `try` block, something quite different happens: the call stack is "unwound" by removing the calls to the functions "below" the one that contains the `try` statement. Any of the ***`try-statements`*** that haven't yet executed are abandoned. Execution continues with the statements in the `except` clause, and then moves on to the statement that follows the entire `try/except` statement. Figures 4-6 and 4-7 show the difference.

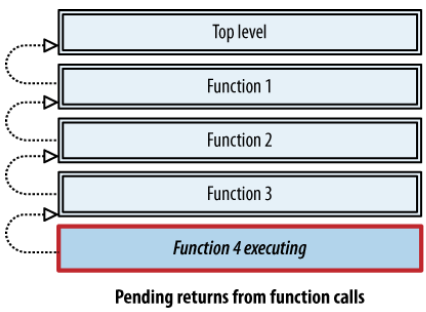


Figure 4-6. Pending returns from function calls

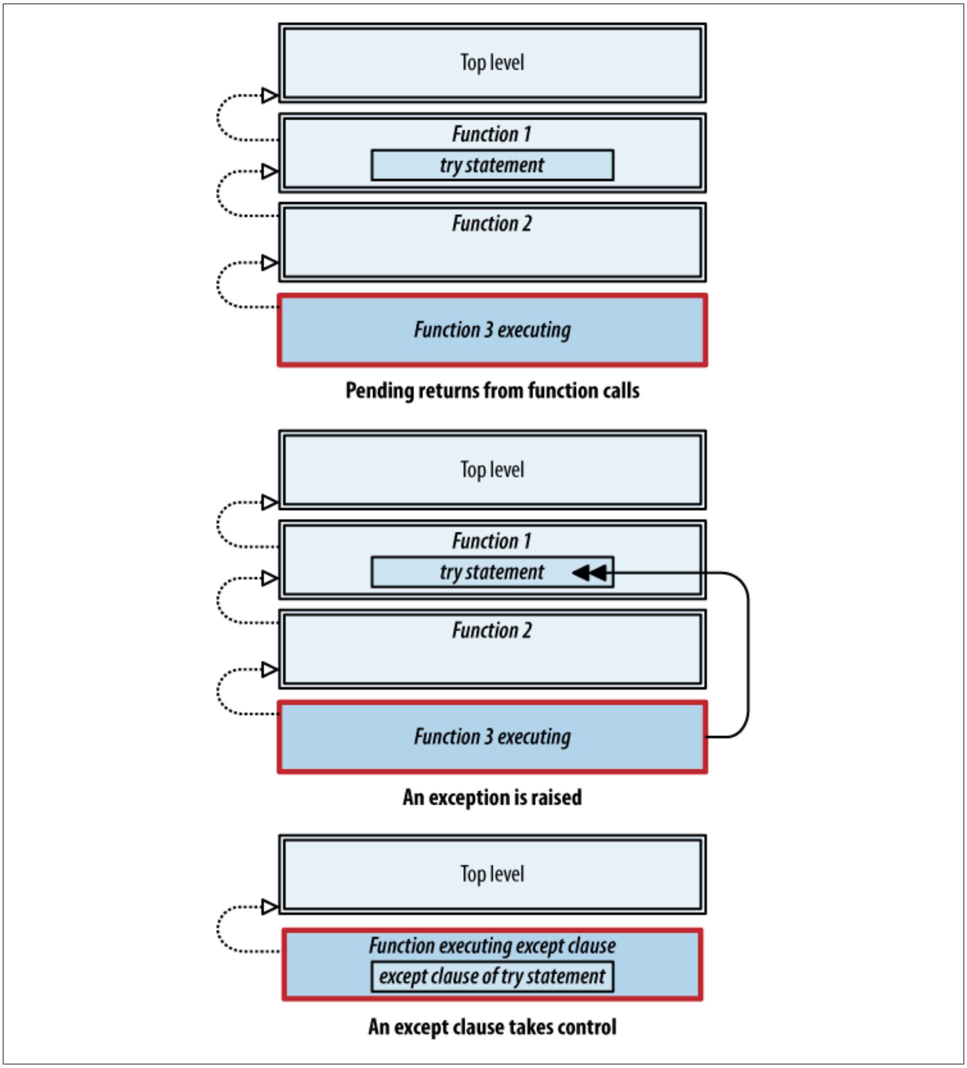


Figure 4-7. Pending returns from function calls with an exception

Optional features of exception handling statements

The `try` statement offers quite a few options. The difficulty here is not so much in comprehending all the details, although that does take some time. The real challenge is to develop a concrete picture of how control "flows" through function calls and Python's various kinds of statements. Then you can begin developing an understanding of the very different flow induced by exceptions.

STATEMENT

Exception Handling Options

`try` statement options include:

- Binding a name to the exception object caught by an `except` clause by following the exception class name with `as` and a name, allowing the statements in the clause to use details from the exception instance
- Multiple `except` clauses, each naming a different condition
- `except` clauses that specify a list of error types instead of just one
- A final `except` clause with no exception class to catch any exception not caught in one of the other `except` clauses
- A `finally` clause whose statements are always executed, regardless of whether an error occurs—in fact, the statements in a `finally` clause are executed even if the `try` clause or an `except` clause executes a `return`

Here's a template that shows the use of all of these features together:

```
try:
    statements
except ErrorClass1:
    statements1
except (ErrorClass2, ErrorClass3):
    statements2
except ErrorClass4 as err:
    statements that can refer to err
except:
    statements that are executed if an error occurs
    whose type is not in one of the above except clauses
finally:
    statements that always get executed, whether or not an error occurs
```

Now that we know how to handle errors, what changes might we want to make in our little program for finding IDs? Suppose we've decided we want the program to print an error message whenever it fails to open a file, but then continue with the next one. This is easily accomplished with one simple `try` statement:

```
def get_gi_ids(filename):
    try:
        with open(filename) as file:
            return [extract_gi_id(line) for line in file
                                        if line[0] == '>']
    except IOError:
        print('File', filename, 'not found or not readable.')
        return []
```

Note that the `except` clause returns an empty list rather than returning `None` or allowing the function to end without a `return` (which amounts to the same thing). This is because the function that calls this one will be concatenating the result with a list it is accumulating, and since `None` isn't a sequence it can't be added to one. (That's another

TypeError you'll often see, usually as a result of forgetting to return a value from a function.) If you've named the exception with `as` *name*, you can `print(`*name*`)` instead of or in addition to your own message.

Incidentally, this `with` statement:

```
with open('filename') as file:
    use file
```

is roughly the same as:

```
try:
    file = open('filename')
    use file
finally:
file.close()
```

The `finally` clause guarantees that the file will be closed whether an error happens or not—and without the programmer having to remember to close it. This is a great convenience that avoids several kinds of common problems. The `with` statement requires only one line in addition to the statements to be executed, rather than the four lines required by the `try` version.

Exception handling and generator objects

An important special use of `try` statements is with generator objects. Each call to `next` with a generator object as its argument produces the generator's next value. When there are no more values, `next` returns the value of its optional second argument, if one is provided. If not, a `StopIteration` error is raised.

There are two ways to use `next`: either you can provide a default value and compare it to the value `next` returns each time, or you can omit the argument and put the call to `next` inside a `try` that has an `except StopIteration` clause. (An `except` clause with no exception class or a `finally` would also catch the error.)

An advantage of the exception approach is that the `try` statement that catches it can be several function calls back; also, you don't have to check the value returned by `next` each time. This is particularly useful when one function calls another that calls another, and so on. A one-argument call to `next` in the innermost function and a `try` statement in the top-level function will terminate the entire process and hand control back to the top-level function, which catches `StopIteration`.

Raising Exceptions

Exception raising isn't limited to library functions—your code can raise them too.

The raise statement

The `raise` statement is used to raise an exception and initiate exception handling.

STATEMENT

Exception Raising

The `raise` statement creates an instance of an exception class and begins the exception handling process.

```
raise exception-expression
```

The *`exception-expression`* can be any expression whose value is either an exception class or an instance of one. If it is an exception class, the statement simply creates an instance for you. Creating your own instance allows you to specify arguments to the new instance—typically a message providing more detail about the condition encountered. The class `Exception` can be used for whatever purposes you want, and it can take an arbitrary number of arguments. You can put statements like this in your code:

```
raise Exception('File does not appear to be in FASTA format.', filename)
```

The statements in any of a `try` statement's exception clauses can "reraise" an exception using a `raise` statement with no expression. In that case, the stack unwinding resumes and continues until the next `try` statement prepared to handle the exception is encountered.

Not only can your code raise exceptions, but you can create your own exception classes and raise instances of those. (The next chapter shows you how to create your own classes.) It's especially important for people building modules for other people to use, since code in a module has no way of knowing what code from the outside wants to do when various kinds of problems are encountered. The only reasonable thing to do is design modules to define appropriate exception classes and document them for users of the module so they know what exceptions their code should be prepared to handle.

Raising an exception to end a loop

The point was made earlier that exceptions aren't necessarily errors. You can use a combination of `try` and `raise` statements as an alternative way of ending loops. You would do this if you had written a long sequence of functions that call each other, expecting certain kinds of values in return. When something fails deep down in a sequence of calls it can be very awkward to return `None` or some other failure value back through the series of callers, as each of them would have to test the value(s) it got back to see whether it should continue or itself return `None`. A common example is repeatedly using `str.find` in many different functions to work through a large string.

Using exception handling, you can write code without all that distracting error reporting and checking. Exceptional situations can be handled by raising an error. The first function called can have a "while-true" loop inside a `try` statement. Whenever some function determines that nothing remains to process, it can throw an exception. A good exception class for this purpose is `StopIteration`, which is used in the implementation of generators, `while-as` statements, and other mechanisms we've seen:

```
try:
    while(True):
        begin complicated multi-function input processing
except StopIteration:
    pass

... many definitions of functions that call each other; ...
... wherever one detects the end of input, it does: ...
        raise StopIteration
```

Extended Examples

This section presents some extended examples that make use of the constructs described earlier in the chapter.

Extracting Information from an HTML File

Our first example in this section is based on the technique just discussed of raising an exception to end the processing of some text. Consider how you would go about extracting information from a complex HTML page. For example, go to NCBI's Entrez Gene site (*http://www.ncbi.nlm.nih.gov/sites/entrez*), enter a gene name in the search field, click the search button, and then save the page as an HTML file. Our example uses the gene vWF.# Example 4-29 shows a program for extracting some information from the results returned. The patterns it uses are very specific to results saved from Entrez Gene, but the program would be quite useful if you needed to process many such pages.

Example 4-29. Searching for data in an HTML file

```
endresults = '- - - - - - - - end Results - - - - - -'
patterns = ('</em>]',
            '\n',
            '</a></div><div class="rprtMainSec"><div class="summary">',
            )

def get_field(contents, pattern, endpos):
    endpos = contents.rfind(pattern, 0, endpos)
    if endpos < 0:
        raise StopIteration
    startpos = contents.rfind('>', 0, endpos)
    return (endpos, contents[startpos+1:endpos])

def get_next(contents, endpos):
    fields = []
    for pattern in patterns:
        endpos, field = get_field(contents, pattern, endpos)
```

#vWF stands for "von Willebrand Factor," which plays a role in von Willebrand disease, the most common human hereditary coagulation abnormality. There are several forms of the disease, other genes involved, and complex hereditary patterns.

```
        fields.append(field)
    fields.reverse()
    return endpos, fields

def get_gene_info(contents):
    lst = []
    endpos = contents.rfind(endresults, 0, len(contents))
    try:
        while(True):
            endpos, fields = get_next(contents, endpos)
            lst.append(fields)
    except StopIteration:
        pass
    lst.reverse()
    return lst

def get_gene_info_from_file(filename):
    with open(filename) as file:
        contents = file.read()
    return get_gene_info(contents)

def show_gene_info_from_file(filename):
    infolst = get_gene_info_from_file(filename)
    for info in infolst:
        print(info[0], info[1], info[2], sep='\n    ')

if __name__ == '__main__':
    show_gene_info_from_file(sys.argv[1]
                       if len(sys.argv) > 1
                       else 'EntrezGeneResults.html')
```

Output for the first page of the Entrez Gene results for vWF looks like this:

```
Vwf
    Von Willebrand factor homolog
    Mus musculus
VWF
    von Willebrand factor
    Homo sapiens
VWF
    von Willebrand factor
    Canis lupus familiaris
Vwf
    Von Willebrand factor homolog
    Rattus norvegicus
VWF
    von Willebrand factor
    Bos taurus
VWF
    von Willebrand factor
    Pan troglodytes
VWF
    von Willebrand factor
    Macaca mulatta
```

```
vwf
    von Willebrand factor
    Danio rerio
VWF
    von Willebrand factor
    Gallus gallus
VWF
    von Willebrand factor
    Sus scrofa
Vwf
    lectin
    Bombyx mori
VWF
    von Willebrand factor
    Oryctolagus cuniculus
VWF
    von Willebrand factor
    Felis catus
VWF
    von Willebrand factor
    Monodelphis domestica
VWFL2
    von Willebrand Factor like 2
    Ciona intestinalis
ADAMTS13
    ADAM metallopeptidase with thrombospondin type 1 motif, 13
    Homo sapiens
MADE_03506
    Secreted protein, containing von Willebrand factor (vWF) type A domain
    Alteromonas macleodii 'Deep ecotype'
NOR51B_705
    putative secreted protein, containing von Willebrand factor (vWF) type A domain
    gamma proteobacterium NOR51-B
BLD_1637
    von Willebrand factor (vWF) domain containing protein
    Bifidobacterium longum DJO10A
NOR53_416
    secreted protein, containing von Willebrand factor (vWF) type A domain
    gamma proteobacterium NOR5-3
```

This code was developed in stages. The first version of the program had separate functions `get_symbol`, `get_name`, and `get_species`. Once they were cleaned up and working correctly it became obvious that they each did the same thing, just with a different pattern. They were therefore replaced with a single function that had an additional parameter for the search pattern.

The original definition of `get_next` contained repetitious lines. This definition replaces those with an iteration over a list of patterns. These changes made the whole program easily extensible. To extract more fields, we just have to add appropriate search patterns to the `patterns` list.

It should also be noted that because the second line of some entries showed an "Official Symbol" and "Name" but others didn't, it turned out to be easier to search backward

from the end of the file. The first step is to find the line demarcating the end of the results. Then the file contents are searched in reverse for each pattern in turn, from the beginning of the file to where the last search left off. (Note that although you might expect it to be the other way around, the arguments to `rfind` are interpreted just like the arguments to `find`, with the second less than the third.)

The Grand Unified Bioinformatics File Parser

This section explores some ways the process of reading information from text files can be generalized.

Reading the sequences in a FASTA file

Example 4-30 presents a set of functions for reading the sequences in a FASTA file. They are actually quite general, and can work for a variety of the kinds of formats typically seen in bioinformatics. The code is a lot like what we've seen in earlier examples. All that is needed to make these functions work for a specific file format is an appropriate definition of `skip_intro` and `next_item`.

Example 4-30. Reading the sequences in a FASTA file

```
def get_items_from_file(filename, testfn=None):
    """Return all the items in the file named filename; if testfn
    then include only those items for which testfn is true"""
    with open(filename) as file:
        return get_items(file, testfn)

def find_item_in_file(filename, testfn=None):
    """Return the first item in the file named filename; if testfn
    then return the first item for which testfn is true"""
    with open(filename) as file:
        return find_item(file, testfn)

def find_item(src, testfn):
    """Return the first item in src; if testfn then return the first item for which testfn is true"""
    gen = item_generator(src, testfn)
    item = next(gen)
    if not testfn:
        return item
    else:
        try:
            while not testfn(item):
                item = next(gen)
            return item
        except StopIteration:
            return None

def get_items(src, testfn=None):
    """Return all the items in src; if testfn then include
    only those items for which testfn is true"""
    return [item for item in item_generator(src)
```

```
            if not testfn or testfn(item)]

def item_generator(src):
    """Return a generator that produces a FASTA sequence from src each time it is called"""
    skip_intro(src)
    seq = ''
    description = src.readline().split('|')
    line = src.readline()
    while line:
        while line and line[0] != '>':
            seq += line
            line = src.readline()
        yield (description, seq)
        seq = ''
        description = line.split('|')
        line = src.readline()

def skip_intro(src):
    """Skip introductory text that appears in src before the first item"""
    pass                                # no introduction in a FASTA file
```

The functions `get_items_from_file` and `find_item_in_file` simply take a filename and call `get_items` and `find_item`, respectively. If you already have an open file, you can pass it directly to `get_items` or `find_item`. All four functions take an optional filter function. If one is provided, only items for which the function returns true are included. Typically, a filter function like this would be a lambda expression. Note that `find_item` can be called repeatedly on the same open file, returning the next item for which `testfn` is true, because after the first one is found the rest of the source is still available for reading.

`next_item` is a generator version of the functions we've seen for reading FASTA entries. It reads one entry each time it is called, returning the split description line and the sequence as a pair. This function and possibly `skip_intro` would need to be defined differently for different file formats. The other four functions stay the same.

Generalized parsing

Extracting a structured representation from a text file is known as *parsing*. Python, for example, parses text typed at the interpreter prompt or imported from a module in order to convert it into an executable representation according to the language's rules. Much of bioinformatics programming involves parsing files in a wide variety of formats. Despite the ways that formats differ, programs to parse them have a substantial underlying similarity, as reflected in the following template.

TEMPLATE

Grand Unified Bioinformatics File Parser

Example 4-30 defines six functions. Four of them are essentially universal, but each file format will require its own definition of `next_item`, and if the format includes introductory text that must be skipped, `skip_intro` will need to be redefined as well. A large proportion of bioinformatics text files can be read using this set of six functions. As a reference, an outline of what each does follows.

```
# ---- Convenience functions for starting with a filename ----
get_items_from_file(filename, testfn=None)
find_item_in_file(filenamee, testfn=None):

# ---- Primary functions: get all and find next ----
get_items(src, testfn=None)
find_item(src, testfn=None)

# ---- Format-specific functions ----
skip_intro(src)
next_item(src)
```

Parsing GenBank Files

Next, we'll look at an example of applying the generalized parser template to read features and sequences from GenBank flat files.* There are many ways to navigate in a browser to get a page in GenBank format from the NCBI website.† For instance, if you know the GenInfo Identifier (GI), you can get to the corresponding GenBank record using the URL *http://www.ncbi.nlm.nih.gov/nuccore/* followed by the GI number. Then, to download the page as a flat text file, simply click on the "Download" drop-down on the right side of the page just above the name of the sequence and select "GenBank" as the format. The file will be downloaded as *sequence.gb* to your browser's default download directory.

There's a great deal of information in these GenBank entries. For this example we just want to extract the accession code, GI number, feature information, and sequence. Example 4-31 shows the code needed to implement the format-specific part of the unified parser template: `skip_intro` and `next_item`. For a given format, the implementation of either of these two functions may require other supporting functions.

* See *http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html* for an example and an explanation.

† See *http://www.ncbi.nlm.nih.gov*.

Example 4-31. Reading sequences from a GenBank file

```
def get_GenBank_items_and_sequence_from_file(filename):
    with open(filename) as file:
        return [get_ids(file), get_items(file), get_sequence(file)]

def get_ids(src):
    line = src.readline()
    while not line.startswith('VERSION'):
        line = src.readline()
    parts = line.split()                        # split at whitespace; removes \n
    assert 3 == len(parts), parts               # should be VERSION acc GI:id
    giparts = parts[2].partition(':')
    assert giparts[2], giparts                  # if no colon, [1] & [2] are empty
    assert giparts[2].isdigit()                 # all numbers?
    return (parts[1], giparts[2])

def get_sequence(src):
    """Return the DNA sequence found at end of src"""
    # When this is called the ORIGIN line should have just been read,
    # so we just have to read the sequence lines until the // at the end
    seq = ''
    line = src.readline()
    while not line.startswith('//'):
        seq += line[10:-1].replace(' ', '')
        line = src.readline()
    return seq

def skip_intro(src):
    """Skip introductory text that appears before the first item in src"""
    line = src.readline()
    while not line.startswith('FEATURES'):
        line = src.readline()

attribute_prefix = 21*' ' + '/'
def is_attribute_start(line):
    return line and line.startswith(attribute_prefix)

def is_feature_start(line):
    return line and line[5] != ' '

def next_item(src):
    """Return a generator that produces a FASTA sequence from src each time it is called"""
    skip_intro(src)
    line = src.readline()
    while not line.startswith('ORIGIN'):
        assert is_feature_start(line)           # line should start a feature
        feature, line =  read_feature(src, line)
        # need to keep line to feed back to read_feature
        yield feature
```

```
def read_feature(src, line):
    feature = line.split()
    props = {}
    line = src.readline()
    while not is_feature_start(line):
        key, value = line.strip()[1:].split('=')
        # remove initial / and split into [feature, value]
        if value[0] == '"':
            value = value[1:]                        # remove first "; remove final " later
        fullvalue, line = read_value(src, line, value)
        # need to keep line to feed back to read_value
        props[key] = fullvalue
    feature.append(props)
    return feature, line

def read_value(src, line, value):
    line = src.readline()
    while (not is_attribute_start(line) and
           not is_feature_start(line)):
        value += line.strip()
        line = src.readline()
    if value[-1] == '"':
        value = value[:-1]                           # remove final "
    return value, line
```

The template is meant as a helpful outline, not a restrictive structure. A program written according to this template may add its own actions. In this case, the "items" to be read are the features. Before reading the features, the program extracts the accession and GI numbers. After all the features have been read, an extra step is needed to read the sequence. The top-level function returns those items in a list: a pair containing the accession and GI numbers, the list of features, and the sequence. Each feature is a list containing the type of the feature, the range of bases it covers, and a dictionary of key/value pairs defining properties.

For the GenBank sample record saved from *http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html*, the result of executing the code would be as follows (sequence strings have been truncated to fit on a single line, and explanations have been added to the output as comments):

```
>>> data = get_genbank_items_and_sequence_from_file('sequences/sample.gb')
>>> pprint.pprint(data)
[('U49845.1', '1293613'),                        # (accession, GI)
 [['source',                                     # first feature is source
   '1..5028',                                    # range of source within base seq
   {'chromosome': 'IX',
    'db_xref': 'taxon:4932',                     # reference to Taxonomy DB
    'map': '9',
    'organism': 'Saccharomyces cerevisiae'}],
  ['CDS',                                        # coding sequence
   '<1..206',                                    # seq from base 1 to 206, 5' partial
   {'codon_start': '3',                          # translation starts at 3
    'db_xref': 'GI:1293614',                     # protein GI
    'product': 'TCP1-beta',                      # protein produced by the CDS
```

```
    'protein_id': 'AAA98665.1',                     # protein accession
    'translation': 'SSIYNGIS...'}],                 # amino acid sequence
                                          # gene AXL2 spans nucleotides 687 through 3158
  ['gene', '687..3158', {'gene': 'AXL2'}],
  ['CDS',
   '687..3158',                                     # a complete coding sequence
   {'codon_start': '1',
    'db_xref': 'GI:1293615',
    'function': 'required for axial budding pattern of S.cerevisiae',
    'gene': 'AXL2',
    'note': 'plasma membrane glycoprotein',
    'product': 'Axl2p',
    'protein_id': 'AAA98666.1',
    'translation': 'MTQLQISL...'}],
  ['gene', 'complement(3300..4037)', {'gene': 'REV7'}],
  ['CDS',
   'complement(3300..4037)',                        # CDS is on opposite strand
   {'codon_start': '1',
    'db_xref': 'GI:1293616',
    'gene': 'REV7',
    'product': 'Rev7p',
    'protein_id': 'AAA98667.1',
    'translation': 'MNRWVEKW...'}]],
                                                    # base sequence:
 'gatcctccatatacaacggtatctccacctcaggtttagatctcaacaacggaaccattgcc...']
```

Translating RNA Sequences

Next, we're going to build a program to translate RNA sequences directly into strings of three-letter amino acid abbreviations. We actually got a start on this in the previous chapter, where we defined a codon table and a lookup function. They will be repeated here for convenience.

Step 1

Example 4-32 begins a series of function definitions interleaved with brief explanatory text and sample printouts.

Example 4-32. Translating RNA sequences, step 1

```
RNA_codon_table = {
#                        Second Base
#        U             C             A             G
# U
    'UUU': 'Phe', 'UCU': 'Ser', 'UAU': 'Tyr', 'UGU': 'Cys',     # UxU
    'UUC': 'Phe', 'UCC': 'Ser', 'UAC': 'Tyr', 'UGC': 'Cys',     # UxC
    'UUA': 'Leu', 'UCA': 'Ser', 'UAA': '---', 'UGA': '---',     # UxA
    'UUG': 'Leu', 'UCG': 'Ser', 'UAG': '---', 'UGG': 'Trp',     # UxG
# C
    'CUU': 'Leu', 'CCU': 'Pro', 'CAU': 'His', 'CGU': 'Arg',     # CxU
    'CUC': 'Leu', 'CCC': 'Pro', 'CAC': 'His', 'CGC': 'Arg',     # CxC
    'CUA': 'Leu', 'CCA': 'Pro', 'CAA': 'Gln', 'CGA': 'Arg',     # CxA
    'CUG': 'Leu', 'CCG': 'Pro', 'CAG': 'Gln', 'CGG': 'Arg',     # CxG
# A
    'AUU': 'Ile', 'ACU': 'Thr', 'AAU': 'Asn', 'AGU': 'Ser',     # AxU
    'AUC': 'Ile', 'ACC': 'Thr', 'AAC': 'Asn', 'AGC': 'Ser',     # AxC
    'AUA': 'Ile', 'ACA': 'Thr', 'AAA': 'Lys', 'AGA': 'Arg',     # AxA
    'AUG': 'Met', 'ACG': 'Thr', 'AAG': 'Lys', 'AGG': 'Arg',     # AxG
# G
    'GUU': 'Val', 'GCU': 'Ala', 'GAU': 'Asp', 'GGU': 'Gly',     # GxU
    'GUC': 'Val', 'GCC': 'Ala', 'GAC': 'Asp', 'GGC': 'Gly',     # GxC
    'GUA': 'Val', 'GCA': 'Ala', 'GAA': 'Glu', 'GGA': 'Gly',     # GxA
    'GUG': 'Val', 'GCG': 'Ala', 'GAG': 'Glu', 'GGG': 'Gly'      # GxG
}

def translate_RNA_codon(codon):
    return RNA_codon_table[codon]
```

Step 2

The next step is to write a function that translates an RNA base string into a string of the corresponding three-letter amino acid abbreviations. The optional ***step*** argument to `range` is useful for this. Testing with assertions while this code was being developed revealed the need to ignore the last base or two of sequences whose length is not a multiple of 3, something not considered when the code was first written. The expression `len(seq)%3` gives the remainder when the length of the sequence is divided by 3—we have to subtract that from `len(seq)` so we don't try to process an excess base or two at the end of the sequence. The new example is shown in Example 4-33.

Example 4-33. Translating RNA sequences, step 2

```
def translate(seq):
    """Return the animo acid sequence corresponding to the RNA sequence seq"""
    translation = ''
    for n in range(0, len(seq) - (len(seq) % 3), 3):              # every third base
        translation += translate_RNA_codon(seq[n:n+3])
    return translation
```

Step 3

Next, we take care of frame shifts and add printing functions with the functions shown in Example 4-34.

Example 4-34. Translating RNA sequences, step 3

```
def translate_in_frame(seq, framenum):
    """Return the translation of seq in framenum 1, 2, or 3"""
    return translate(seq[framenum-1:])

def print_translation_in_frame(seq, framenum, prefix):
    """Print the translation of seq in framenum preceded by prefix"""
        print(prefix,
          framenum,
          ' ' * framenum,
          translate_in_frame(seq, framenum),
          sep='')

def print_translations(seq, prefix=''):
    """Print the translations of seq in all three reading frames, each preceded by prefix"""
    print('\n' ,' ' * (len(prefix) + 2), seq, sep='')
    for framenum in range(1,4):
        print_translation_in_frame(seq, framenum, prefix)

>>> print_translations('ATGCGTGAGGCTCTCAA')
  ATGCGTGAGGCTCTCAA
1 MetArgGluAlaLeu
2  CysValArgLeuSer
3   Ala---GlySerGln
>>> print_translations('ATGATATGGAGGAGGTAGCCGCGCGCCATGCGCGCTATATTTTGGTAT')
  ATGATATGGAGGAGGTAGCCGCGCGCCATGCGCGCTATATTTTGGTAT
1 MetIleTrpArgArg---ProArgAlaMetArgAlaIlePheTrpTyr
2  ---TyrGlyGlyGlySerArgAlaProCysAlaLeuTyrPheGly
3   AspMetGluGluValAlaAlaArgHisAlaArgTyrIleLeuVal
```

Step 4

Now we are ready to find the open reading frames. (We make the simplifying assumption that we're using the standard genetic code.) The second and third functions here are essentially the same as in the previous step, except that they call `translate_with_open_reading_frames` instead of `translate_in_frame`. Example 4-35 shows the new definitions.

Example 4-35. Translating RNA sequences, step 4

```
def translate_with_open_reading_frames(seq, framenum):
    """Return the translation of seq in framenum (1, 2, or 3), with ---'s when not within an
    open reading frame; assume the read is not in an open frame when at the beginning of seq"""
    open = False
    translation = ""
    seqlength = len(seq) - (framenum - 1)
    for n in range(frame-1, seqlength - (seqlength % 3), 3):
        codon = translate_RNA_codon(seq[n:n+3])
```

```
        open = (open or codon == "Met") and not (codon == "---")
        translation += codon if open else "---"
    return translation

def print_translation_with_open_reading_frame(seq, framenum, prefix):
    print(prefix,
          framenum,
          ' ' * framenum,
          translate_with_open_reading_frames(seq, framenum),
          sep='')

def print_translations_with_open_reading_frames(seq, prefix=''):
    print('\n', ' ' * (len(prefix) + 2), seq, sep='')
    for frame in range(1,4):
        print_translation_with_open_reading_frame(seq, frame, prefix)

>>> print_translations('ATGCGTGAGGCTCTCAA')
  ATGCGTGAGGCTCTCAA
1 MetArgGluAlaLeu
2  ---------------
3   ---------------
>>> print_translations('ATGATATGGAGGAGGTAGCCGCGCGCCATGCGCGCTATATTTTGGTAT')
  ATGATATGGAGGAGGTAGCCGCGCGCCATGCGCGCTATATTTTGGTAT
1 MetIleTrpArgArg------------MetArgAlaIlePheTrpTyr
2  ---------------------------------------------
3   ---MetGluGluValAlaAlaArgHisAlaArgTyrIleLeuVal
```

Step 5

Finally, we print the sequence both forward and backward. Getting the reverse of a sequence is easy, even though there's no function for it: `seq[::-1]`. Remember that trick, as you will need it any time you want to reverse a string. Working with biological sequence data, that will be quite often! Example 4-36 shows the final piece of the code.

Example 4-36. Translating RNA sequences, step 5

```
def print_translations_in_frames_in_both_directions(seq):
    print_translations(seq, 'FRF')
    print_translations(seq[::-1], 'RRF')

def print_translations_with_open_reading_frames_in_both_directions(seq):
    print_translations_with_open_reading_frames(seq, 'FRF')
    print_translations_with_open_reading_frames(seq[::-1], 'RRF')

>>> print_translations('ATGCGTGAGGCTCTCAA')
     ATGCGTGAGGCTCTCAA
FRF1 MetArgGluAlaLeu
FRF2  ---------------
FRF3   ---------------

     AACTCTCGGAGTGCGTA
RRF1 ---------------
RRF2  ---------------
RRF3   ---------------
```

```
>>> print_translations('ATGATATGGAGGAGGTAGCCGCGCGCCATGCGCGCTATATTTTGGTAT')
     ATGATATGGAGGAGGTAGCCGCGCGCCATGCGCGCTATATTTTGGTAT
FRF1 MetIleTrpArgArg------------MetArgAlaIlePheTrpTyr
FRF2  ---------------------------------------------
FRF3   ---MetGluGluValAlaAlaArgHisAlaArgTyrIleLeuVal

     TATGGTTTTATATCGCGCGTACCGCGCGCCGATGGAGGAGGTATAGTA
RRF1 ------------------------------------------------
RRF2  MetValLeuTyrArgAlaTyrArgAlaProMetGluGluVal---
RRF3   ---------------------------------------------
```

Constructing a Table from a Text File

Our next project will be to construct a table from a text file. We'll use the file located at *http://rebase.neb.com/rebase/link_bionet*, which contains a list of restriction enzymes and their cut sites. The file has a simple format.‡ Each line of data has the form:

```
EnzymeName (Prototype)       ... spaces ...    CutSite
```

A "prototype" in this data set is the first enzyme to be discovered with the specified cut site. Lines that represent prototypes do not have anything in the "Prototype" column.

Several lines of information appear at the beginning of the file, before the actual data. To ensure that our program ignores these lines we'll use one of our usual `skip_intro`-type functions, here called `get_first_line`. A look at the file shows that the first line of data is the first one that begins with an A. This is certainly not an acceptable approach for "production" software, since the organization of the file might change in the future, but it's good enough for this example. The end of the file may have some blank lines, and we'll need to ignore those too.

To represent the data in this file we'll construct a dictionary whose keys are enzyme names and whose values are the cut sites. We'll make this simple and ignore the information about prototypes. Because there are so many details we are going to take things a step at a time. This is how you should work on your programs too.

Step 1

The general outline of the program will be:

1. Initialize the enzyme table.
2. Skip introductory lines, returning the first real line.
3. While line is not empty:
    a. Parse line.
    b. Store entry in the enzyme table.
    c. Read another line.

‡ The site contains files in other formats with more information. We'll use one of those later in the book.

Turning those steps into function names and adding as few details as we can get away with, we write some essentially empty functions (see Example 4-37). For example, `get_first_line` just returns an empty string; in a sense it's done its job, which is to return a `line`.

Example 4-37. Simple Rebase reader, step 1

```
def load_enzyme_table():
    return load_enzyme_data_into_table({})
    # start with empty dictionary

def load_enzyme_data_into_table(table):
    line = get_first_line()
    while not end_of_data(line):
        parse(line)
        store_entry(table)
        line = get_next_line()
    return table

def get_first_line():
    return ''                                  # stop immediately

def get_next_line():
    return ' '                                 # so it stops after getting the first line

def end_of_data(line):
    return True

def parse(line):
    return line

def store_entry(table):
    pass

# testing:
def test():
    table = load_enzyme_table()
    assert len(table) == 0
    print('All tests passed.')

test()
```

Step 2

We can fill in the details of some of these functions immediately. Do not be disturbed that some of the definitions remain trivial even after all the changes we'll make. We might want to modify this program to use with a different file format, and there's no guarantee that `get_next_line`, for instance, will always be as simple as it is here. Using all these function names makes it very clear what the code is doing without having to comment it.

In the following steps, changes from and additions to the previous step are highlighted. They include:

- Binding names to the result of `parse` (using tuple unpacking)
- Passing the key and value obtained from `parse` to `store_entry`
- Returning a result from `first_line` to use in testing
- Implementing `end_of_data`
- Splitting the line into fields and returning the first and last using tuple packing
- Implementing `store_entry`

Many of the functions continue to have "pretend" implementations in step 2, which is shown in Example 4-38.

Example 4-38. Simple Rebase reader, step 2

```
def load_enzyme_table():
    return load_enzyme_data_into_table({})
    # start with empty dictionary

def load_enzyme_data_into_table(table)
    line = get_first_line()
    while not end_of_data(line):
        key, value = parse(line)
        store_entry(table, key, value)
        line = get_next_line()
    return table

def get_first_line():
    return 'enzymeA (protoA)             CCCGGG'
    # return a typical line

def get_next_line():
    return ' '                          # so it stops after getting the first line

def end_of_data(line):
    return len(line) < 2
                                        # 0 means end of file, 1 would be a blank line

def parse(line):
    fields = line.split()
                                        # with no argument, split splits at whitespace
                                        # tuple packing (omitting optional parens)
    return fields[0], fields[-1]
                                        # avoiding having to determine whether there are 2 or 3

def store_entry(table, key, value):
    table[key] = value

def test():
    table = load_enzyme_table()
    assert len(table) == 1
    result =  parse('enzymeA (protoA)             CCCGGG')
```

```
    assert result == ('enzymeA', 'CCCGGG'), result
    print('All tests passed.')

test()
```

Step 3

In the next step, we actually read from the file. It's silly to try to wrestle with a large datafile while you are writing the functions to handle it. Instead, extract a small bit of the actual file and put it in a "test file" that you can use until the program seems to work. We'll construct a file called *rebase_test_data01.txt* that contains exactly the following text:

```
some introductory text
more introductory text
AnEnzyme (APrototype)             cutsite1
APrototype                        cutsite2
```

Changes in this step include making some of the definitions more realistic:

- Adding a filename parameter to `load_enzyme_table`
- Embedding most of that function in a loop
- Adding a call to `print` for debugging purposes
- Passing the open file object to `return_first_line` and `get_next_line`
- Implementing `get_first_line` and `get_next_line`

Example 4-39 illustrates the third step.

Example 4-39. Simple Rebase reader, step 3

```
def load_enzyme_table(data_filename):
    with open(data_filename) as datafile:
        return load_enzyme_data_into_table(datafile, {})

def load_enzyme_data_into_table(datafile, table)
    line = get_first_line(datafile)
    while not end_of_data(line):
        print(line, end='')
        key, value = parse(line)
        store_entry(table, key, value)
        line = get_next_line(datafile)
    return table

def get_first_line(fil):
     line = fil.readline()
     while line and not line[0] == 'A':
         line = fil.readline()
     return line

def get_next_line(fil):
    return fil.readline()
```

```
def end_of_data(line):
    return len(line) < 2

def parse(line):
    fields = line.split()
    return fields[0], fields[-1]

def store_entry(table, key, value):
    table[key] = value

def test():
    print()
    datafilename = 'rebase_test_data01.txt'
    table = load_enzyme_table(datafilename)
    assert len(table) == 2, table
    result =  parse('enzymeA (protoA)            CCCGGG')
    assert result == ('enzymeA', 'CCCGGG'), result
    print()
    print('All tests passed.')

test()
```

Step 4

Finally, we clean up some of the code (not shown here), use the real file, and test some results. Example 4-40 shows step 4.

Example 4-40. Simple Rebase reader, step 4

```
# This step uses the definitions of the previous step unchanged, except
# that the call to print in load_enzyme_data_into_table could be removed

def test():
    print()
    datafilename = 'link_bionet.txt'
    table = load_enzyme_table(datafilename)
    # check first entry from file:
    assert table['AaaI'] == 'C^GGCCG'
    # check an ordinary entry with a prototype:
    assert table['AbaI'] == 'T^GATCA', table
    # check an ordinary entry that is a prototype:
    assert table['BclI'] == 'T^GATCA', table
    # check last entry from file:
    assert table['Zsp2I'] == 'ATGCA^T'
    assert len(table) == 3559, len(table)
    print()
    print('All tests passed.')
```

Step 5

If we wanted to, we could also add a function for printing the table in a simpler format to make it easier to read in the future and a corresponding function for doing that reading. We can print each entry of the table on a separate line with the name of the

enzyme separated by a tab from the sequence it recognizes. Example 4-41 shows these two simple functions.

Example 4-41. Simple Rebase reader, step 5

```
def write_table_to_filename(table, data_filename):
    """Write table in a simple format to a file named data_filename"""
    with open(data_filename, 'w') as file:
        write_table_entries(table, files)

def write_table_entries(table, datafile):
    for enzyme in sorted(table.keys()):
        print(enzyme, table[enzyme], sep='        ', file=datafile)

def read_table_from_filename(data_filename):
    """Return a table read from the file named data_filename
    that was previously written by write_table_to_filename"""
    with open(data_filename) as file:
        return read_table_entries(file, {})

def read_table_entries(datafile):
    for line in datafile:
        fields = line.split()
        table[fields[0]] = fields[1]
    return table
```

Tips, Traps, and Tracebacks

Tips

- Keep functions small.
- Prefer comprehensions to loops or iterations that collect values.
- Look for opportunities to use conditional expressions—`(val1 if val2 else val3)`—instead of `if`/`else` statements when the two clauses contain just an expression or assign the same name.
- The mechanisms discussed in this chapter are the core of Python programs. You should review the chapter from time to time to better understand parts you didn't completely follow the first time through.
- Use the templates as a reference—they capture a large portion of the ways control statements are used.
- In general, if a function is to return `True` if a certain condition holds and `False` otherwise, spelling out the following is technically "silly":

  ```
  if condition:
      return True
  else:
      return False
  ```

Instead, just write `return` *`condition`*. For example, instead of:

```
if len(seq1) > len(seq2):
    return True
else:
    return False
```

write:

```
return len(seq1) > len(seq2)
```

The result of the comparison *is* `True` or `False.`

- When it is last in the function, a statement such as the following (without an `else` clause):

  ```
  if boolean-condition:
      return expression
  ```

  can be more concisely written as:

  ```
  return boolean-condition and expression
  ```

  This expression will return `False` if *`condition`* is false and the value of *`expression`* if it is true. The primary reason you would need to use the conditional statement instead of *`expression`* is if you specifically want the function to return `None` if *`condition`* is false, rather than `False`.

- While assertions are valuable for testing code and for checking conditions at certain points in function definitions while the definitions are still under development, do not use assertions as a coding technique in place of conditionals. For example, if you want to check the validity of some input, do that in a conditional statement and take appropriate action if the condition is violated, such as raising an error. Do not simply assert that condition.[§]

- The first Python code in a file should be a docstring. If enclosed in triple single or double quotes, the docstring can be as many lines long as you want. If your file is imported (rather than executed), the module that will be created for it will store that docstring, and Python's help facility will be able to display it. You'll probably want to comment the file to explain what it is, even if only for your own future reference, so you might as well get in the habit of using docstrings rather than comments.

- You should make a choice about how you will use single single, single double, triple single, and triple double quotes, then follow that convention in a consistent way. The reason for using each of the four kinds of quotes in a different way is that it makes it easier to search for appearances in your code of a string that serves a particular purpose, such as a docstring.

§ Other than coding style, the issue here is that assertions are meant to be executed only during development, not in a "production" version of a program. When running Python from the command line, the `-O` option can be added to optimize a few aspects of the execution, one of which is to ignore assertions.

The following choices were made for the code in this book and the downloadable code files:

Single single quotes
: Used for short strings

Single double quotes
: Used for short strings with internal single quotes

Triple double quotes
: Used for docstrings

Triple single quotes
: Used for long value strings

The difference between single single quotes and single double quotes is not so important, but it's better to use one most of the time. Don't forget that you can include a single quote inside a single-quoted string (or a double quote inside a double-quoted string) simply by putting a backslash before it.

- If a compilation error points to a line that appears to be OK, the problem is probably on the previous line. Check that all "structural" lines of compound statements—`def`, `if`, `else`, `elif`, `while`, `for`, `try`, `except`, and `finally`—end with a colon.
- When your IDE indents a line to an unexpected level, don't fight it or ignore it—it's giving you very useful information about a problem in your code. Make sure that each colon in a compound statement is at the end of its "logical line" (i.e., that the beginning and any continuation lines are treated as one). The only keywords that are followed by a colon (and therefore nothing else on that line) are `else`, `try`, and `finally`.
- Here's a useful debugging function. It is better than just using `assert` for testing code because an assertion failure stops the program, whereas failures identified by calls to this function do not:

```
def expect_equal(expected, result):
    """Return expected == result, printing an error message if not true;
    use by itself or with assert in a statement such as:
        assert expect_equal(3438,
                            count_hypothetical_proteins(gbk_filename)"""
    if expected == result:
        return True
    print('Expected', expected, 'but got', result)
```

Traps

- Many functions that collect values, whether using a loop or an iteration, must avoid adding to the collection on the first pass through the loop or iteration. When that is the case, the loop or iteration must be followed by a statement that adds the last item to the collection.

- When processing data read from outside the program, *do not trust* that all of it is in the expected form. There may be missing or aberrant lines or entries; for example, an enzyme cut site that should be `?` to mean "unknown" may be the digit `7` instead.
- It is rarely a good idea to process large files by calling `readlines`. Rather, process one line at a time, by calling `readline` in a loop, by using a comprehension over the lines of the file, or by using a `for` statement. This avoids creating enormous lists that occupy large amounts of memory.

Tracebacks

Following are some representative error messages:

`AttributeError: 'range' object has no attribute 'index'`
: A call has been made to a method not supported by the type.

`EOFError`
: The programmer (or user) typed Ctrl-D (Ctrl-Z on Windows).

`IndentationError: unexpected indent`
: This can happen (well) after a `try` statement with no `except` or `finally` clause.

`IOError: [Errno 2] No such file or directory: 'aa2.fasta'`
: There are a number of `IOError` variations, each with a different `Errno` and message format.

`KeyboardInterrupt`
: The user typed Ctrl-C twice.

`KeyError`
: An attempt has been made to reference a dictionary element by a key not present in the dictionary.

`TypeError: unorderable types int() < str()`
: An attempt has been made to compare two values of different types.

`TypeError: 'NoneType' object is not iterable`
: The code contains a `for` statement that is iterating over something that has no value. This is *not* the same as iterating over an empty collection, which, although it does nothing, does not cause an error. Errors that mention `NoneType` are almost always caused by a function meant to return a value that does not include a `return` statement.

`object is not iterable: 'builtin_function_or_method'`
: A function name was used in place of a function call in the `in` part of a comprehension or `for` statement; i.e., you forgot the parentheses. This is a common mistake when calling `dict.keys()`, `dict.values()`, and `dict.items()`.

CHAPTER 5
Classes

Simple collections are appropriate for organizing relatively simple and transient information. Elaborately nested collections, though, are not. Consider the output parsed from the very small GenBank entry shown in Example 4-31. Even though it ignores much of the file's information, the program produces an elaborate nested structure. Working with such a structure creates the following serious problems:

Complexity
: Using the representation in a program requires understanding its intricacies.

Awkward navigation
: Accessing specific parts of the representation requires tricky combinations of index expressions and function calls.

Exposure to change
: Changes to the representation require widespread changes in every program that uses it.

The combination of these factors causes mental strain, slows programming, makes debugging more difficult, and increases mistakes. It is too easy to write `[1]` instead of `[0]`, or `[1]` instead of `[0][1]`. Later, such index expressions have no apparent meaning, so they are difficult to understand and correct. If changes must be made it is difficult to make them consistently and find every place that must be edited. While collections are powerful tools, they are best reserved for actions that operate on the entire collection, rather than being used with access expressions or functions that pick out specific elements.

These problems arise because of the entanglement of code that does need to know about representation details and code that does not. For instance, we can define a function called `get_organism` that extracts the name of the organism from the complex collection. That function needs to know the details of the representation. However, other functions can just call `get_organism` when they need that piece of the collection. With this approach, if the structure of the collection is changed in a way that affects

the way an organism's name is accessed, only the `get_organism` function will be affected, not any of the code that calls it.

Functions like `get_organism` hide details from code that uses a representation. Only a small number of functions need to work with the representation directly; everything else calls those functions. This strategy defines a *functional interface* to the complex structure. Changes to the representation cause changes only to the small number of functions in the interface. This enforces the principle of *information hiding* that dictates the *separation of implementation and use*.

We have already seen two kinds of information hiding:

- Accessing built-in Python types through functions and methods without knowing anything about how they are represented
- Dividing code into small chunks, with higher-level functions that organize the computation separated from lower-level ones dealing with the details of file formats and collection structures

The comments in the preceding paragraphs demonstrate the need for another kind:

- Packaging together a representation along with the functions that manipulate it so that code can call those functions rather than accessing the data

If we had the third kind of information hiding, the result would be much like the first. What we need, then, is a way to define new *types* (*classes*) the way we can define new *functions*. Python does indeed give us such a mechanism.

Defining Classes

Python's collection types work as follows:

1. The type is called like a function to create new instances.
2. The call to the type can include arguments used to initialize the new instance.
3. Methods are invoked on instances of types instead of accessing their internal representations.
4. Methods defined for different types can have the same name: when a method is called through a value, Python finds the appropriate function by looking in the definition of the value's type.
5. The way an instance is printed is based on its type.

None of this is special to collection types. These features are intrinsic to Python's type mechanisms. They are what make Python an *object-oriented* programming language.

Python's class definition statement defines a new class (type), just the way a definition statement defines a new function. The keyword that introduces the statement is `class`. Like function definition statements, class definition statements are one-clause compound statements containing other statements. The other statements are nearly

always function definitions, but any statement is allowed. Each `def` statement inside a class defines a method for the instances of that class.

New instances are created by calling the name of the class as a function, just as with built-in types. Each instance stores information in named *fields*. Together, fields and methods are called *attributes*. They are accessed with the usual dot notation, such that this expression:

```
obj.name()
```

would call the `name` method defined by `obj`'s type, and this expression:

```
obj.species
```

would access the `species` attribute of `obj`. That attribute might be a field or it might be a method. Using `str` as an example:

```
>>> 'AAU'.isupper                        # ACCESS the isupper ATTRIBUTE
<built-in method isupper of str object at 0x1c523e0>
>>> 'AAU'.isupper()                      # CALL the isupper METHOD
True
```

What classes should your programs define? They can define pretty much whatever you can name. Classes add nouns to the language you are building with your code, the way functions add verbs. The definition of a class defines the functional interface to the data its instances represent. Classes are the mechanism that supports information hiding.

Collections serve two very different purposes: *grouping* and *structuring*. A group has no intrinsic meaning—it just collects values together. A complex collection structure is different. The data returned by the GenBank parser of Example 4-31 (see "Parsing GenBank Files" on page 148) is a list, but it's not the same kind of list as one containing primitive values. It has many levels of structure, with many elements playing specific roles in what the collection represents. Classes should be used in place of all but the simplest structuring collections.

STATEMENT

Class Definition

A class definition begins with the keyword `class` and a name, followed by a colon and one or more statements indented relative to the first line.

```
class name:

    def method1(args...):
        . . .
    def method2(args...):
        . . .

    . . .
```

You should begin each syllable of your class names with an uppercase letter. Omit the underscores that would otherwise separate syllables in the usual Python naming convention.

The statement in Example 5-1 defines a new Python type called `GenBankEntry`. Using `pass` for the class body is just like using `pass` for the body of a function you know you want to define but haven't yet determined how to implement.

Example 5-1. Defining a class

```
class GenBankEntry:
    pass

>>> GenBankEntry()
<__main__.GenBankEntry object at 0x1cafdb0>
>>> GenBankEntry()
<__main__.GenBankEntry object at 0x1caff30>
```

The interpreter's output when a new instance of `GenBankEntry` is created shows the name of its module (`__main__`, indicating the interpreter), the name of the class, and a hexadecimal memory address. The memory addresses don't mean anything, but they at least show that we have created two different instances of the class.

Values may be stored in fields of an instance using dot notation. We'll do this initially for demonstration purposes only—in general, we'll want to use a functional interface:

```
>>> gb = GenBankEntry()
>>> gb.accession = 'U49845.1'
>>> gb.gi = '1293613'
>>> gb.accession
'U49845.1'
```

Instance Attributes

To demonstrate important features of classes, instances, and methods, we'll build a class to represent the information parsed from a GenBank entry as shown in Example 4-31. The code will grow rather long as more methods are added, so only the new methods are shown for each example.

Access methods

The first kind of methods we will consider are *access methods*, which return values based on the fields of an instance. The simplest just return the value of a field. We want programs using our class to call methods with expressions, such as `gb.get_gi()`, instead of directly accessing the fields with expressions such as `gb.gi`. To support that, we need to define the method `get_gi()` inside the class. It will return the value of the `gi` field for the instance on which it is called.

How does a method determine which instance's fields to access? Python automatically adds the instance through which a method is called to the beginning of the call's argument list. Therefore, the instance a method uses is the value of its first parameter. By convention, the name of that parameter is `self`:

```
class GenBankEntry:

    def get_gi(self):
        return self.gi

    def get_accession(self):
        return self.accession

>>> gb.get_accession()
'U49845.1'
```

Suppose we decide to change the way we store the accession and GI values. We might want to move them into a dictionary called `info`, anticipating the addition of more keys and values. It would then be necessary to redefine the methods based on this new representation:

```
class GenBankEntry:

    info = {}

    def get_gi(self):
        return self.info['gi']

    def get_accession(self):
        return self.info['accession']
```

Any code that uses the access functions would be completely unaffected by this change. This is the essence of information hiding—separating the implementation of a representation from its use:

```
>>> gb = GenBankEntry()
>>> gb.info = {'accession': 'U49845.1', 'gi': '1293613'}
>>> gb.get_accession()
'U49845.1'
```

Now suppose we extract the source information from the data returned by the GenBank file parser and store it in a field called `source`. The source is the first feature in the data's list of features, which itself is the second element of the list returned. We'll store the source in one field and a list of the rest of the features in another:

```
>>> data = get_GenBank_items_and_sequence_from_file(genbank_file_name)
                  # defined in Example 4-31
>>> gb.source = data[1][0]
>>> gb.features = data[1][1:]
>>> pprint.pprint(gb.source)
['source',
 '1..5028',
 {'chromosome': 'IX',
  'db_xref': 'taxon:4932',
```

```
 'map': '9',
 'organism': 'Saccharomyces cerevisiae'}]
```

We don't want outside code digging around in the source's qualifier dictionary, or even knowing where it is and that it *is* a dictionary. So, we'll define access methods to get values from the qualifier dictionary. We will *not* define a method that returns the value of the `source` field itself, since that would expose the representation of that part of the information:

```
class GenBankEntry:

    def organism(self):
        return self.source[2].get('organism', None)

    def chromosome(self):
        return self.source[2].get('chromosome', None)
```

Not all access methods return data that is directly stored in the instance. Some perform numerical computations, string manipulations, or other operations using the stored data:

```
class GenBankEntry:

    def gc_content(self):
        return round((100 * ((self.sequence.count('g') +
                              self.sequence.count('c'))) /
                      len(self.sequence)),

                     2)
>>> gb.gc_content()
37.97
```

Other methods do some kind of filtering or searching to find the desired information. It isn't even necessary for the data a method filters or searches to be the value of one of the instance's fields; for example, a method might search a collection obtained by calling an access method of a different instance. Here is the definition of one that filters and extracts the instance's feature list to produce a list of genes and the range of their coding region(s):

```
    def genes(self):
        return [(feature[1], feature[2]['gene'])
                for feature in self.features
                if feature[0] == 'gene']

>>> gb.genes()
[('687..3158', 'AXL2'), ('complement(3300..4037)', 'REV7')]
```

TEMPLATE

Access Methods

Access methods return values based on the fields of an instance. They can do much more than just return the value of a field. Part of information hiding, in fact, is that

programmers using access methods don't have to know which ones refer to similarly named fields. Access methods generally perform one of the following kinds of computations:

- *Get* the value of a field
- *Look up* a value in a dictionary that is the value of a field
- *Compute* a value from the value of one or more fields
- *Filter* the collection that is the value of a field
- *Search for* a targeted value in a field that has a collection value or in some other collection outside the instance

Predicate methods

A *predicate* is a Boolean-valued method or function. A predicate method is essentially an access method that also does a comparison. Classes should define methods for predicates that are likely to be important, difficult to compute, or frequently used. It is better to define predicate methods for other code to use than it is to leave the other code to interpret the meaning of values returned from access methods.

To highlight that a method is a predicate, its name should usually begin with `is` (we've already seen quite a few predicates defined by the `str` class, such as `isupper`). Occasionally, however, the grammar of the word(s) used for the method name calls for the prefix `has`. If the rest of the method name has multiple syllables you would normally include an underscore after the prefix, beginning the name with `is_` or `has_`. A test for whether a `GenBankEntry` contains a base sequence or an amino acid sequence would probably be the most useful predicate for our class:

```
class GenBankEntry:

    def is_base_sequence(self):
        return set(self.get_sequence().lower()) <= {'a', 'c', 't', 'g'}
```

One predicate that it is often important to provide is one that determines whether an instance is before another. This would be used to implement some kind of ordering among the instances of the class. The method `__lt__` is called by the built-in function `sorted` and the method `list.sort`. Any call to one of these can include a `key` argument to specify an ad hoc sorting order. If no `key` argument is supplied in a call, `__lt__` is used.

> Method names like `__lt__` that begin and end with two underscores are called *special methods*. They play predefined roles in Python's implementation. You are not allowed to give other methods names that begin and end with two underscores.

Many classes should implement `__lt__`, even if all it does is compare the name or identifier of its two arguments. For a class that does not implement `__lt__`, it is an error to

try to sort a sequence of its instances. The `__lt__` method must have two parameters: `self` and another for the other object to which `self` will be compared. If we want the method to be able to compare only objects of the same type, we can begin its definition by testing whether they are of the same type and raising an exception if not. Here's a definition of `__lt__` that compares two instances of `GenBankEntry` using their GenInfo IDs:

```
class GenBankEntry:

    def __lt__(self, other):
        if type(self) != type(other):
            raise Exception('Incompatible argument to __lt__: ' +
                            str(other))
        return self.get_gid() < other.get_gid()
```

Though limited to comparing two instances of the same type, that definition is general enough to capture it as a template.

TEMPLATE

Supporting Sorting

The method `__lt__` is called by `sorted` and `list.sort` when no `key` argument is provided. A simple definition would check that its two arguments have the same type; more sophisticated definitions could handle multiple types.

```
class ClassName:

    def __lt__(self, other):
        if type(self) != type(other):
            raise Exception(
                'Incompatible argument to __lt__: ' +
                str(other))
        return self.get_something() < other.get_something()
```

Initialization methods

One special method, `__init__`, is critical to the way classes work. When a class is called to create a new instance, the arguments included in the call are passed automatically to the `__init__` method defined in the class. This method can initialize the instance's fields and ensure that the new instance is ready for use. This is how fields are normally assigned their initial values, not by using dot notation with the field name outside of the class.

The `__init__` method plays a fundamental role in the interface defined by the class. Although other methods could add other fields, the norm is to assign all fields in a class's `__init__` method, even if the value is only `None`. To help anyone reading the class, including its author, the `__init__` method should go first in the body of the class's definition.

Following is the definition of a possible `__init__` method for the `GenBankEntry` class. It is designed to take just one argument other than `self`: the full complex of data read from the GenBank entry parser. Note that `__init__` methods embody the choices the class's author has made about field names and what goes in which fields. Because code outside the class would use access methods, not field names, no one using the class needs to know about anything inside the `__init__` method:

```
class GenBankEntry:

    def __init__(self, data):
        self.accession, self.version = data[0][0].split('.')
        self.gid = data[0][1]
        self.sequence = data[2]
        # assuming first feature is source & treating specially
        assert 'source' == data[1][0][0]
        self.features = data[1][1:]
        self.source = data[1][0]
```

Now we no longer need to assign fields outside of the class. Instead, we just call the class with the data obtained from the parser:

```
>>> gb = GenBankEntry(data)
```

This creates a new instance, calls `GenBankEntry.__init__` (passing it the new instance as the first argument and `data` as the second), and assigns `gb` to the result. In this one case, there is no instance in front of a dot in the method call: Python automatically and invisibly calls the `__init__` method as part of the instance creation process. By the time it calls `__init__`, the new instance has been created, so it can be passed as the `self` argument to the method. Note that the `__init__` method does not return a value. After the method is called, the new instance is returned as the value of the call to the class.

String methods

Two other important special methods are `__str__` and `__repr__`. These define how Python prints instances of a class. Each returns a string. What that string contains is up to the class implementer.

The two methods are slightly different, as shown in Table 5-1. If a class defines `__repr__` but not `__str__`, `__repr__` will be called in place of `__str__`. However, if a class defines `__str__` but not `__repr__`, `__str__` will not be called—the default definition of `repr` will be used instead.

Table 5-1. __str__ versus __repr__

	__str__	__repr__
Called by	`str(obj)`	`repr(obj)`
Returns	A human-readable string	A string that could be typed at the interactive prompt or included in a Python file to create an equivalent object
Where "called by" functions are called	`print`	Interpreter output

The "called by" functions call the methods. The interpreter and the `print` function call the "called by" functions to get a string, then print it without the quotes. With neither `__str__` nor `__repr__` defined yet, the four possible outputs would be as follows:

```
>>> gb
<__main__.GenBankEntry object at 0x1c69db0>          # default repr, no quotes
>>> repr(gb)
'<__main__.GenBankEntry object at 0x1c69db0>'        # default repr, quotes
>>> print(gb)
<__main__.GenBankEntry object at 0x1c69db0>          # default repr, no quotes
>>> str(gb)
'<__main__.GenBankEntry object at 0x1c69db0>'        # default repr, quotes
```

Here is a simple definition of `GenBankEntry.__str__`:

```
class GenBankEntry:

    def __str__(self):
        return 'GenBankEntry-' + self.get_gid()
```

With that definition in place, here's what happens with the same four expressions:

```
>>> gb
<__main__.GenBankEntry object at 0x1c69db0>          # default repr, no quotes
>>> repr(gb)
'<__main__.GenBankEntry object at 0x1c69db0>'        # default repr, quotes
>>> print(gb)
GenBankEntry-1293613                                 # __str__, no quotes
>>> str(gb)
'GenBankEntry-1293613'                               # __str__, quotes
```

There are really two kinds of `__repr__` implementations. For classes whose instances can be represented succinctly, such as `range`, the goal is to have the interpreter output be the same as the input:

```
>>> range(1,5)
range(1,5)
```

For other classes, the goal is more pragmatic. We could define the first kind of `__repr__` method for `GenBankEntry`, as it contains all the information needed to reconstruct the complex collection structure that was passed to its initialization method. That wouldn't be a good idea, however: any time a `GenBankEntry` was typed to the interpreter the resulting printout would be very long. That doesn't necessarily mean we must put up with the uninformative default, though. We can define `__repr__` in a way that is convenient for interaction with the interpreter, debugging, and so on.

If we change the name of the `__str__` method to `__repr__`, the same method will be called in all four of our situations, since `__str__` defaults to `__repr__` if not defined:

```
class GenBankEntry:

    def __repr__(self):                              # instead of __str__
        return 'GenBankEntry-' + self.get_gid()
```

This is what happens with the four expressions now:

```
>>> gb
'GenBankEntry-1293613'                              # __repr__, no quotes
>>> repr(gb)
'GenBankEntry-1293613'                              # __repr__, quotes
>>> print(gb)
GenBankEntry-1293613                                # __repr__, no quotes
>>> str(gb)
'GenBankEntry-1293613'                              # __repr__, quotes
```

Finally, we also have the option of defining both methods. It is often useful to define `__repr__` to give more information than `__str__`. Usually you'll want to keep the result of `__str__` fairly compact, since it is called by `print`. Let's define the two methods as follows, then run our expressions again:

```
class GenBankEntry:

    def __str__(self):
        return "GenBankEntry-" + self.get_gid()

    def __repr__(self):
        return "<GenBankEntry {} {} '{}'>".format(self.get_gid(),
                                                  self.get_accession(),
                                                  self.organism()
                                                  )

>>> gb
<GenBankEntry 1293613 U49845.1 'Saccharomyces cerevisiae'>
>>> repr(gb)
"<GenBankEntry 1293613 U49845.1 'Saccharomyces cerevisiae'>"
>>> print(gb)
GenBankEntry-1293613
>>> str(gb)
'GenBankEntry-1293613'
```

> These definitions of `__str__` and `__repr__` do not directly access fields of the instances. Information hiding is just as useful inside a class as outside. Generally, only a small number of methods need direct access to fields; the others can use those. That allows decisions about field names and contents to be changed without requiring extensive editing of the other methods of the class.

Modification methods

In many cases, once an instance has been created many of a class's fields will not be changed. As we've defined it so far, `GenBankEntry` is completely static—none of the values of an instance's fields will ever change. To make it possible for a program to change the value of a field without violating the information hiding principle, the class must define a *modification method* for that field.

Stretching our example just a bit, suppose we are writing a program that manages a number of `GenBankEntry` instances for its user, and we want to add a field to contain

notes the user makes. Since the user may make multiple notations, we'll store them in a list. We need to do three things to implement this:

1. Assign the field to a new list in `__init__`.
2. Define an access method to retrieve the notes.
3. Define a modification method to add a note.

As with access methods, there are different kinds of modification methods that you can use, depending on the type of the field's value. The method may just replace the primitive value of a field, or it may add elements to or remove elements from a field's collection value, navigate into a more complex field value, change multiple fields, and so on. For this example, let's assume that each note is a string, and all that happens when a note is added is that it gets added to the end of a list of previous notes. The implementation looks like this:

```
class GenBankEntry:

    def __init__(self):
        # . . . other statements . . .
        notes = []                              # initialize the notes field

    def get_notes(self):
        return self.notes

    def add_note(self, note):
        self.notes.append(note)

    def remove_note(self, note):
        self.notes.remove(note)
```

Modification methods are important in another way: with all changes to instances going through these methods, they can control exactly what happens before and after the changes are made. Before making any change, a method could check the validity of the new value(s) supplied. After making the changes, it might update associated information. These are all details that should be hidden behind modification methods.

Defining modification methods also has practical advantages for the class implementer. Such methods can include code to print debugging information, or they can log changes to a file for later examination. They are important places to set debugging breakpoints in order to observe changes being made to instances.

Before moving on, we need to address a subtle problem in the definition of `get_notes`. Because it returns the actual list value of the `notes` field, external code can modify that list. The same would be true if a dictionary that is the value of a field were returned. Changes to the list or dictionary might require other actions the external code wouldn't know about, such as printing debugging information or updating information located elsewhere. Even if the external code did perform such actions, changes to the representation or required actions would invalidate that code. As always, we want to protect instances from outside meddling.

The solution, like the problem, is a bit subtle. Access methods should never return the actual collection value of a field. Instead, they should return either copies or a generator. We should therefore define `get_notes` in one of two ways:

```
class GenBankEntry:

    # return copy of list
    def get_notes(self):
        return self.notes[:]

    # return generator of the elements of the list
    def get_notes(self):
        return (note for note in self.notes)
```

Action methods

Some classes, like `GenBankEntry`, are designed simply to *represent* something. Except perhaps for some modification methods and, of course, `__init__`, they are entirely passive; all of their other methods simply return values.

Other classes are more active; they provide methods that *do* something, in the sense of having an effect outside the class. They may perform input and output operations, create instances of other classes, or perform a wide variety of other sorts of activities. Often, the actions are specific to a particular application.

There isn't much to say about action methods, other than to point out the possibility that a class might have them. They are entirely unconstrained in the arguments they take and the computations they perform. We'll see an example in a parser class later in this chapter.

Support methods

Classes are not limited to defining methods for external code to use. We don't want long method definitions any more than we want long function definitions outside of classes. If a method meant for external use grows too long, it can be divided into multiple methods. As with code extracted from long function definitions, it may turn out that these small methods can be used from several others within the class. Normally they would *not* be used from outside of the class.

We will call these methods "support methods." They should be commented as such to indicate that they are not really part of the class's interface.

Summary

This section examined various kinds of methods in piecemeal fashion. It concludes with a template for a basic class definition. Note that because it plays such a fundamental role in defining instance content, we put `__init__` first. Then, since they too are special functions, we put `__repr__` and `__str__` before the other method groups. Next

come predicates, partly to account for the remaining special method—`__lt__`—and partly because predicates are the simplest methods, just returning a Boolean value.

TEMPLATE

Basic Class

This is a high-level outline of the organization of a typical basic class definition. With the exception of the fundamental methods listed first, there may be any number of methods in each category.

```
class ClassName:

# Fundamental Methods

    def __init__(self, ...):
        # initialize new instance's fields
        # no return

    def __repr__(self):
        return # string used by the interpreter to print instances

    def __str__(self):
        return # string used by print and str

# Predicates

    def __lt__(self, other):
        if type(self) != type(other):
            raise Exception(
               'Incompatible argument to __lt__: ' +
               str(other))
        return self.somemethod() < other.somemethod()

    def is_some_characteristic(self):
        return # True or False

# Access Methods

    def get_something(self):
        return # value obtained by one of:
                   # accessing a field
                   # lookup by key
                   # computation
                   # filtering a collection
                   # searching a collection

# Modification Methods

    def set_something(self, ...):
        # change the value of one or more fields based on
        # the parameter values supplied in the call.
        # generally no return value

# Action Methods

    def do_something(self, ...):
        # do something that has effects outside the class
```

```
# Private Support Methods

    def helper_method(self, ...):
        # something used by other methods of the class only
```

Class Attributes

Some fields and methods should be associated with the class itself, rather than its individual instances. Typical uses for *class attributes* are generating a unique ID number for each instance and keeping track of all of a class's instances.

Each class defines its own namespace, just the way modules and functions (including methods) do. That is what allows methods in different classes to have the same name. When a function is defined in a module, its name is added to the module's namespace. When a method is defined in a class, its name is added to the class's namespace.

Actually, "added" is not quite correct: it would be more accurate to say "names and values are added if they're not already there, and otherwise the names are rebound." It is perfectly legitimate, though pointless, to define the same function several times in a row. The first definition adds the function name to its containing namespace, and the others simply rebind what that name means.

Assignment statements also define names, adding (or replacing) names and values within their namespaces. Assignment statements can appear anywhere any other statement can appear. In particular, they can appear inside class definitions alongside the usual method definitions. An assignment statement at the top level of the class definition adds a *class field* to the class's namespace.

Attributes in a class's scope can be accessed in one of two ways:

- As usual, through an instance of the class using dot notation (including `self` inside one of the class's methods)
- Through the class itself, using dot notation

Class fields

Python classes do not track their instances. That is to say, there is no method you can call on a class to get a list of all its instances. However, we can add instance tracking to `GenBankEntry` with only two more lines of code: an assignment statement to initialize the class field and a line in the `__init__` method to add the new instance to its value. We can use a list or a dictionary, depending on whether we want to be able to look up instances by a key. Since it is very likely that we will want to find instances of `GenBankEntry` by their GenInfo IDs, we'll use those as the keys in a dictionary.

We use the same capitalization convention for the names of class fields as for the names of classes themselves. This avoids confusion as to which names are for instance fields and which for class fields.

The following code sets up the instance dictionary:

```
class GenBankEntry:

    Instances = {}                          # Initialize the instance dictionary

    def __init__(self):
        # . . . other statements . . .
        self.Instances[self.gid] = self
        # add the new instance to the class's Instances dictionary
```

After creating the `GenBankEntry` instance from the data we've been using for our examples, we would see one instance in the class's `Instances` dictionary:

```
>>> GenBankEntry.Instances                  # access to field through class
{'1293613': <GenBankEntry 1293613 U49845.1 'Saccharomyces cerevisiae'>}
```

TEMPLATE

Tracking a Class's Instances

It is often useful for a class to track its instances. All that is required is to assign a name to an empty list or dictionary in a top-level assignment statement and add a line to `__init__` to add the new instance to the collection.

```
class UsingList:
    Instances = []
    def __init__(self[, arg, ...]):
        # . . .
        self.Instances.append(self)

class UsingDictionary:
    Instances = {}
    def __init__(self[, arg, ...]):
        # . . .
        self.Instances[self.somefield] = self
```

Another common use for class fields is to keep track of the number of instances that have been generated without tracking the instances themselves. This is useful in order to give each new instance a unique identifier when the class doesn't have a field that can act as one. The identifier would typically be used to define `__str__` and/or `__repr__` methods that distinguish the instances more helpfully than the awkward default implementation of `__repr__`. They can also be used to define `__lt__` if there is nothing else on which to base comparisons for the class.

Why not just track all the instances? Sometimes a program generates many instances of a class during one portion of its code, uses them, then doesn't need them anymore. If they are not tracked, Python can reuse the space they occupy. If they are in a collection in a class, though, Python has no way of knowing if they are still needed. You also might want to track instance counts over multiple executions of a program, which you could do by storing the current count when the program exits and restoring it when the program begins execution later.

Initializing the instance count is easy enough—it's an assignment statement just like the one for tracking the instances. There is one difference, though: there is no way to increment the count that is the value of the class field without using the class's name. Based on the preceding discussion, you might think the following would work:

```
class SimpleClass:
❶    InstanceCount = 0
    def __init__(self):
❷        self.number = self.InstanceCount            # no problem
❸        self.InstanceCount = self.InstanceCount + 1 # big problem
>>> sc = SimpleClass()
>>> sc.number
0                                # as expected
>>> SimpleClass.InstanceCount
0                                # surprise! why not incremented?
>>> sc.InstanceCount
1                                # mystery!!
```

As you can see, though, it doesn't. Here's what's going on in this code:

❶ `SimpleClass.InstanceCount` is bound to `0` when the class is defined.

❷ The `__init__` method binds the field `number` of the newly created instance to the current value of `SimpleClass.InstanceCount`, which is `0`.

❸ The full form of the increment statement was used, rather than `+= 1`, to make what happened more explicit. In any assignment statement, the righthand side is evaluated, then the name(s) on the lefthand side is (are) bound to the resulting value(s). The value of `self.InstanceCount` is still `0`, from having been initialized in step 1, so the value of the righthand side is `1`. Finally, `self.InstanceCount` is bound to `1`.

The issue lies in the meaning of `self.InstanceCount` on the lefthand side of the assignment statement. That name refers to a field in the instance, just as `self.number` does. Python doesn't "understand" that you meant it to refer to the class field. The direct solution to this problem is slightly awkward: use the name of the class explicitly when rebinding the value. We didn't need to do this when modifying a list or dictionary used to track instances, because once the class field was assigned it was never rebound. We'll see a better solution shortly.

Class fields can be used for less dynamic purposes too. There are often names that are specific to the class. These could be assigned at the top level of a module but really should go in the class's namespace. Examples include:

- File paths for input and output
- A format string used in `__str__`, `__repr__`, and additional output methods
- Minimum and maximum values

One sophisticated use of class fields turns the assignment problem we just looked at into an advantage. As we saw, once a method assigns a field name in an instance, that name refers to the instance's field, not the class's. The value of a name is determined first by looking for a field by that name in the instance, then in its class. This means that a class field can be used as a default value for instances that don't have their own values for a field with that name. Instances that need a different value than the default can assign their own field with that name. The default can even be changed later in the class, and all instances in which the field has not been assigned will take on the new default value. Any instances that did provide a value for the field will keep that value.

Class methods

The benefits of information hiding are important for class fields too. Are they lists, dictionaries, or more complex collections? Maybe their values are instances of other classes the application defines. Changing the contents of a class field's value may require other changes, and as usual, we don't want outside code dealing with these details. We want to be able to define methods to manage class fields the way other methods manage instance fields.

A *class method* is a method for which `self` is the class itself, not one of its instances. Like a class field, a class method can be accessed using dot notation through either an instance of the class or the class itself.

> The syllables that make up the name of a class method, including the first, should be capitalized, as they are in names of classes and class fields.

How are class methods distinguished from instance methods? A special indicator, `@classmethod`, is placed on the line before a class method definition. (This odd syntax is actually part of a sophisticated aspect of Python that is not discussed further in this book. It is easy enough to use `@classmethod` without learning anything else about `@` notation.) For example:

```
class GenBankEntry:

    Instances = {}                          # initialize the instance dictionary

    @classmethod                            # make the method a class method
    def InstanceCount(self):
        return len(self.Instances)
```

```
    @classmethod
    def GetInstances(self):                    # returning generator as discussed earlier
        return (value for value in self.Instance.keys())

    @classmethod
    def Get(self, target):
        """Return the instance whose GID is target"""
        return self.Instances.get(target, None)
```

Or, if the class keeps its instances in a list instead of a dictionary:

```
class GenBankEntry:

    Instances = []

    @classmethod
    def InstanceCount(self):
        return len(self.Instances)

    @classmethod
    def GetInstances(self):                    # returning generator as discussed earlier
        return (value for value in self.Instances)

    @classmethod
    def Get(self, target):
        """Return the instance whose GID is target"""
        for inst in self.GetInstances:         # using the Find template
            if target == inst.get_gid():
            return inst
```

Earlier, we encountered problems where an `__init__` method incremented a class field used for an instance counter. The solution there was to access the counter through the class name. This is inflexible, though: if the class name is changed, the `__init__` method must be changed too. (There is also a more important reason why this is not a good solution, as we'll see later.) A better option is to use a class method. The `__init__` method shouldn't be reassigning class fields any more than external code should be reassigning instance fields.

An assignment statement in a class method assigns a name in the class's namespace. Assignment statements in a class definition are executed when the `class` statement is executed, and assignment statements in a class method are executed when the method is executed. The following template shows how a counter field should really be implemented.

TEMPLATE

Tracking the Instance Count

This template shows how to track instance counts using a class field. It doesn't show what to do with the `number` field each instance is assigned, but presumably it would be used in the implementation of one or more of `__str__`, `__repr__`, and `__lt__`.

```
class SomeClass:
    InstanceCount = 0                       # initial assignment

    @classmethod
    def IncrementCount(self):
        self.InstanceCount += 1             # reassignment

    def __init__(self, args)
        #...
        self.number = self.InstanceCount
                                            # assign an instance field to
                                            # the value of a class field
        self.IncrementCount()               # class method call
```

Note that the only variable part of the entire template is the name of the class. (Of course, the `number` instance field could also be given a different name.) The template's code can be used exactly as shown in any class.

Classes as objects

We've already discussed class fields and class methods with specific examples and explained their use. There is nothing special or mysterious about class fields and methods. Python treats fields and methods of classes and fields and methods of their instances the same way. This is because *classes are themselves objects*. Remembering this will make it easier to understand how things work.

Class fields and methods are essentially the same as names and functions defined in modules outside of classes. Module functions and values are referenced using dot notation, with the module in front of the dot. Likewise, class fields and methods are referenced using dot notation with the class in front of the dot. A class method can use `self` before the dot; code outside the class would use the name of the class.

An instance method can call both instance and class methods through `self`. A field accessed through `self` in an instance method could be either an instance field or a class field. Python looks first in the instance to see if it contains a field with that name. If it doesn't find it there, it looks in the class. We can't tell by looking at a method call `self.methodname()` or an expression `self.fieldname` whether these refer to attributes of the instance or of its class. That is one reason why it is a good idea to use capitalization to distinguish class attribute names from instance attribute names.

An assignment to `self.fieldname` in an instance method *always* gives `fieldname` a value within the instance, whether or not it already has one in the class. Table 5-2 captures all the possibilities.

Table 5-2. Functions and assignments

	Module m	Class C	Instance t
Internal	**Inside m**	**Inside C**	
Function definition	`def` *fn*`():`	`@classmethod` `def` *Cmeth*`(self)...`	`def` *imeth*`(self)...`
Internal call	*fn*`()`	`self.`*Cmeth*`()`	`self.`*imeth*`()` or `self.`*Cmeth*`()`
Top-level assignment	*name* = *value* Binds *m.name*	*Name* = *value* Binds *C.Name*	*C*`.`*name* = *value*
Assignment inside a function *fn* (like *m* unless preceded by `self`, *C*, or `t`)	*name* = *value* Binds *name* in *fn*	`self.`*Name* = *value* binds *C*`.`*Name*	`self.`*name* = *value* Binds `t.`*name*
Scope order (where Python looks) from inside a function (like *m* unless preceded by `self`, *C*, or `t`)	*name* Function (local) Module Global	`self.`*Name* Class	`self.`*name* Instance Class
External access	**From outside *m***	**From outside *C***	
External call	*m*`.`*name*`()`	*C*`.`*name*`()`	*t*`.`*name*`()`
External field	*m*`.`*name*	*C*`.`*name*	*t*`.`*name*

Compare the module and class columns of the table. With the exception of the use of `self`, there is no difference between the two. Classes and modules are both objects, and their mechanisms are similar. The only difference between the two columns, other than the class's use of `self`, is that the top-level assignment to a class field does *not* use `self`—`self` is only a method parameter, not something associated with the class itself. Note that if a name inside a method is not preceded by `self`, the sequence of scopes in which Python looks for names is the same as if the method had been an ordinary function outside the class.

Comparing the class and instance columns, there are only a few small differences:

- Class method definitions are preceded by `@classmethod`.
- Methods called through `self` in an instance method can be either instance or class methods, whereas methods called through `self` in a class method are always class methods.
- To assign a class field, an instance method must use the class name, exactly as if it were a function outside the class. (It is much better to call a class method that does the assignment so that the name of the class is not part of the method definition.)
- When the value of a field is accessed—not assigned—through `self` in an instance method, Python looks first in the instance and then in the class.

Class and Method Relationships

There are two fundamental ways that multiple classes can be related to each other. The first implements part/whole relationships among the objects of the classes. The second is as a taxonomy of types. They are both hierarchical relationships, but they differ in their implementation and consequences.

Decomposition

Just as it is important to keep functions focused on relatively narrowly defined tasks, it is important to keep classes focused on relatively narrow representations. When function definitions grow uncomfortably large and perform too many actions, they should be broken up into smaller functions that work together. The same is true for classes: when there is too much information in a class, and—often as a consequence—an uncomfortably large number of methods, it is best to break up the class into several that work together.

In addition to being easier to edit and debug, more narrowly focused functions and classes often turn out to be useful beyond their original intended purposes. This happens especially frequently in a field like bioinformatics, where applications often share modules of functions and classes that implement common kinds of data and operations. A class that buries many different kinds of information inside its instances makes it difficult for an application to use only some of that content. In particular, the application might have only some of the data necessary for initializing the instances of the full class.

Dividing up function and class definitions is a programming technique called *decomposition*. This term refers to the act of breaking up the functions and classes, but it also refers to a design strategy. Decomposing large function and class definitions is good technique. Even better, though, is to think through the likely shape of a part of a program in advance and plan to implement a hierarchy of functions and possibly a group of related classes. Becoming adept at this advanced decomposition takes some practice, so you shouldn't be discouraged if your early efforts in this direction don't seem to accomplish much.

Class decomposition

We know how to divide up a function definition into smaller ones that call each other. But how do we divide up a class? The basic principle is the same one that led to creating a class in the first place: complex structuring collections (as opposed to simple grouping collections) are better represented as instances of a class. We therefore look for fields whose values are complex collections and methods that reach into them. Instances of the new class replace the complex collections in instances of the original class.

Let's take a look at the definition of GenBankEntry. Even though it omits a substantial part of the data found in a GenBank file, it still contains a great deal of information. Its definition would include a large number of access methods that reach into collection-valued fields to pull out information. The value of features is a list in which each feature is itself a rather elaborate data structure. The situation calls for defining a separate GenBankFeature class and having the value of fields be a list of GenBankFeature instances.

This is an example of turning a collection of complex collections into a collection of instances. The source field is an example of a field whose value is a single complex collection that could be replaced by a single instance. We separated out source from the other features because it plays a different, and more fundamental, role in the GenBankEntry. Since its value has the same structure as the individual features list, it too can be replaced by an instance of GenBankFeature.

A GenBankFeature class would look something like the definition shown in Example 5-2.

Example 5-2. Definition of GenBankFeature

```
class GenBankFeature:

    # interesting use of a class field!
    FeatureNameOrder = ('gene', 'promoter', 'RBS', 'CDS')

# Fundamental Methods

    def __init__(self, feature_type, locus, qualifiers):
        self.type = feature_type
        self.locus = locus
        self.qualifiers = qualifiers

    def __repr__(self):
        """Return a true __repr__ string"""
        return ('GenBankFeature' +
                repr((self.type, self.locus, self.qualifiers)))
               # repr of the list of arguments __init__ would take

    def __str__(self):
        return self.type + '@' + self.locus

# Predicates

    def __lt__(self, other):
        if type(self) != type(other):
            raise Exception('Incompatible argument to __lt__: ' +
                            str(other))
        return self.locus_lt(other) and self.type_lt(other)

    def is_gene(self):
        return 'gene' == self.get_type()

    def is_cds(self):
        return 'CDS' == self.get_type()
```

```
# Access Methods

    def get_type(self):
        return type

    def get_locus(self):
        return locus

    def get_qualifier(self, name):
        return self.qualifiers.get(name, None)

# Private Support Methods

    def locus_lt(self, other):
        """Is this instance's locus "less than" that of other? This could be quite complicated.
        This is just a simple demonstration that only looks for the first integer."""
        assert type(self) == type(other)      # insisting, not testing
        return ((extract_first_integer(self.get_locus()) or -1) <
                (extract_first_integer(other.get_locus()) or -1))
        # extract_first_integer defined elsewhere; not a method

    def type_lt(self, other):
        assert type(self) == type(other)      # insisting, not testing
        return (self.FeatureNameOrder.find(self.get_type()) <
                other.FeatureNameOrder.find(self.get_type()))
```

Note that `GenBankFeature.__init__` is defined with individual parameters for each of the values it needs, rather than one collection containing all of them. This is actually a more common approach than just providing a single complex collection argument for the method to disassemble. We'll change `GenBankEntry.__init__` similarly, so that we can pass in the `GenBankFeature` instances after they've been created:

```
class GenBankEntry:

     def __init__(self, accession, gid, source, sequence, features):
        # . . .
```

Now we define a pair of functions to take the parsed data and create first the `GenBankFeature` instances and then the `GenBankEntry` itself:

```
def create_genbank_entry(entry_data):
    return GenBankEntry(entry_data[0][0],
                        entry_data[0][1],
                        GenBankFeature(entry_data[1][0]),       # source
                        entry_data[2],
                        [GenBankFeature(info)                   # features
                         for info in entry_data[1][1:]])
```

Separating `GenBankFeature` from `GenBankEntry` doesn't mean that we can't define `GenBankEntry` methods that do things with `GenBankFeature`s. Such `GenBankEntry` methods will basically just call the corresponding methods in `GenBankFeature`. Since the `features` field in a `GenBankEntry` instance is a collection of `GenBankFeature`s, accessing a specific attribute of each of the features would be implemented as a comprehension.

For example:

```
class GenBankEntry:

    def gene_names(self):
        return [feature.get_qualifier('gene') for feature in features
                if feature.is_gene()]
```

This definition doesn't know how to tell if a feature is a gene, so it asks each feature whether it is by using the predicate `is_gene`. Then it collects the names of all features that are genes and returns the resulting collection as its value. When a field's value is not a collection, methods like this can be even simpler. They often just return the result of calling a method on the value of the field. Having replaced `source` with an instance of `GenBankFeature`, we can redefine `GenBankEntry.get_organism` as:

```
class GenBankEntry:

    def get_organism(self):
        return self.source.get_qualifier('organism')
```

Implementing a method primarily by just calling a similar method on a different object is known as *delegation*.

Method decomposition

Since methods are functions, we can apply the usual functional decomposition strategies to break large methods into smaller ones. Instance methods offer a tactical twist that is sometimes convenient and occasionally powerful: methods can use instance fields to share values among related methods instead of passing them as parameters.

We'll collect the GenBank file parser code into a class, to see how values can be shared as instance fields. While we are rewriting this, we can move the code that creates the `GenBankFeature` and `GenBankEntry` instances into the new class, as there is no need for the complex data structure the old code produced or the top-level functions that created instances with them. In doing so, we can compact the code somewhat since we don't need all the features of the original code.

We can also simplify the code by using instance fields to share state among the methods. The code for the parser from Chapter 4 (in "Parsing GenBank Files" on page 148) showed some interesting techniques that are sometimes necessary but that we won't need when implementing the parser as a class. One is that nearly all the functions take an open file as one of their arguments. We can assign an instance field to the open file to be accessed by any method, thereby avoiding having to pass it from one function to the next.

Another situation addressed in that example was that certain functions stopped when they encountered a line meeting some kind of test, but other functions needed that line. As a result, some functions returned the most recently read line, and the functions that called them passed that line as an argument to others. This was one way of "maintaining

the state of the process." Instead of passing the line back and forth among the functions, though, we can just use an instance field.

Example 5-3 outlines the structure of a parser class for GenBank files in terms of which methods would call which. A number of predicates that check whether the current line signals the start or end of some section of the file are included. These predicates can be thought of as testing the state of the parser as it moves through the file. They have been italicized in the figure to distinguish them from the more substantial methods. The method `read_next_line` is omitted from the outline; it is called from 10 places in the code and simply assigns the instance variable `line` to the result of a call to `readline`.

Example 5-3. Using an instance field to share state

```
parse
    get_ids
        is_at_version
    make_feature_generator
        skip_intro
            is_at_features
        is_at_sequence_start
        read_feature
        read_qualifiers
            is_at_feature_start
            read_qualifier_value
                is_at_attribute_start
                is_at_feature_start
        GenBankFeature
    get_sequence
        is_at_sequence_end
    GenBankEntry
```

The definition of a `GenBankParser` class is shown in Example 5-4. The method `parse` parses the file and returns a `GenBankEntry`, including its `GenBankFeature`s. The filename is a parameter to `parse`, and there are no arguments for `__init__`. (An alternative design would be to pass the filename to `__init__` and then call `parse` with no arguments.)

Example 5-4. A GenBankEntry parser class

```
"""Parse a GenBankEntry file, returning a GenBankEntry instance"""

from genbank import GenBankEntry, GenBankFeature

class GenBankParser:

    def __init__(self):
        self.line = None
        self.src = None

    # interesting use of a class field!
    AttributePrefix = (21 * ' ') + '/'
```

```
# Predicates

    def is_at_version(self):
        return self.line and self.line.startswith('VERSION')

    def is_at_features(self):
        return self.line and self.line.startswith('FEATURES')

    def is_at_attribute_start(self):
        return self.line and self.line.startswith(self.AttributePrefix)

    def is_at_feature_start(self):
        return len(self.line) > 5 and self.line[5] != ' '

    def is_at_sequence_start(self):
        return self.line and self.line.startswith('ORIGIN')

    def is_at_sequence_end(self):
        return self.line and self.line.startswith('//')

# Action

    def parse(self, filename):
        """Use the data in file named filename to create and return an instance of GenBankEntry"""
        with open(filename) as self.src:
            accession, gid = self.get_ids()
            feature_generator = self.make_feature_generator()
            source = next(feature_generator)
            features = list(feature_generator)        # remainder after source
            seq = self.get_sequence()
            return GenBankEntry(accession, gid, source, features, seq)

# Action Support

    def read_next_line(self):
        """Better to call this than write its one line of code"""
        self.line = self.src.readline()               # no need to return, just assign

    def get_ids(self):
        """Return the accession and GenInfo IDs"""
        self.read_next_line()
        while not self.is_at_version():
            self.read_next_line()
        parts = self.line.split()
        giparts = parts[2].partition(':')
        return parts[1], giparts[2]

    def skip_intro(self):
        """Skip text that appears in self.src before the first feature"""
        self.read_next_line()
        while not self.is_at_features():
            self.read_next_line()
```

```
def get_sequence(self):
    """Return sequence at end of file"""
    seq = ''
    self.read_next_line()
    while not self.is_at_sequence_end():
        seq += self.line[10:-1].replace(' ', '')
        self.read_next_line()
    return seq

def make_feature_generator(self):
    """Return a generator that produces instances of GenBankFeature
    using the data found in the features section of the file"""
    self.skip_intro()
    while not self.is_at_sequence_start():
        yield self.read_feature()

def read_feature(self):
    """Return an instance of GenBankFeature created
    from the data in the file for a single feature"""
    feature_type, feature_locus = self.line.split()
    self.read_next_line()
    return GenBankFeature(feature_type,
                          feature_locus,
                          self.read_qualifiers())

def read_qualifiers(self):
    qualifiers = {}
    while not self.is_at_feature_start():
        parts = self.line.strip()[1:].split('=')
        key = parts[0]
        value = '' if len(parts) < 2 else parts[1]
        if value and value[0] == '"':
            value = value[1:]                  # remove first quote; last removed later
        qualifiers[key] = self.read_qualifier_value(value)
    return qualifiers

def read__qualifier_value(self, value):
    """With value, the string after a qualifier name, and its equal and quote, keep
    reading lines and removing leading and trailing whitespace, adding to
    value until the final quote is read, then return the accumulated value"""
    self.read_next_line()
    while (not self.is_at_attribute_start() and
           not self.is_at_feature_start()):
        value += self.line.strip()
        self.read_next_line()
    if value and value[-1] == '"':
        value = value[:-1]                     # remove final quote
    return value
```

> It isn't actually necessary for `__init__` to initialize `file` and `line`, the fields that maintain the state of the parsing process. Fields can be assigned for the first time in methods other than `__init__`: until the first assignment of the field is executed, wherever it occurs, the instance just doesn't have a field by that name. However, readers of the code will expect to see all fields initialized in `__init__`, and are unlikely to want to read through the entire class's code checking for field assignments. Therefore, it is good practice to assign all fields the instance will use in the `__init__` method, even if only to `None`. In essence this is a form of documentation, identifying which fields are used by the class's methods.

Even if a method is called from only a few others, it can be useful to assign a field to its arguments. This can be done in the method itself, or it can be done by a calling method before a call, in which case the values will not be included in the call. In fact, even if a method assigns names of its own for local use it can be helpful to turn those into field assignments.

An additional benefit of using this technique is that it makes debugging much easier. The field assignments capture the state of the instance. After the program calls a method on the instance, or while the program is stopped in the debugger, you can look at the values of its fields to figure out what has happened. (The Python debugger is introduced at the end of the next chapter.) This can be easier than tracing or stepping through a series of method calls to watch what values parameters take on.

Singleton classes

Classes that define parsing actions for a particular file format are a common application of object-oriented programming techniques. In this case, the motivations for the definition of a class to parse GenBank entry files were purely technical. Reconfiguring the previous code as a set of methods made certain aspects of the code easier to read, write, and debug, and using instance fields to share state among methods avoided some of the complexity of the functional solution.

To use `GenBankParser`, code would create an instance and call its `parse` method, giving it the name of the file containing the GenBank entry text. That method returns an instance of `GenBankEntry`. Suppose we want to write an application that parses multiple GenBank files. Do we need a new instance of `GenBankParser` for each file? There are two things we need to know to answer that question:

- After `parse` returns, is there any state left in the instance that we'll want to retrieve later?
- After `parse` returns, is there any state left in the instance that would get in the way of using it again?

If the answer to both questions is "no," as it is in this case, we can reuse the same instance to parse multiple files. A different reason for needing only one instance of a

class is that it represents the application itself, the state of the user interface, or something else that must exist during the entire execution of the application. This is different from *reuse*: it is *continual* use.

Classes of which an application needs only one instance are called *singleton classes*. It may seem strange to create a class that has only one instance: most classes are defined in order to have many instances, each with their own state. There's nothing wrong with it, though, and singleton classes are rather common. The concept is introduced here so you'll recognize such classes for what they are.

One unusual thing about singleton classes is that it doesn't matter whether their implementations use all instance fields and methods or all class fields and methods. If we were to copy the definition of `GenBankParser`, delete the `__init__` method, and put `@classmethod` in front of every method, it would work exactly the same as the original definition. The only difference is that instead of creating an instance and calling `parse` on it, we would just call `GenBankParser(`*`filename`*`)` directly.

Sometimes there isn't much reason to choose one approach over another. If there are many methods that might be called from outside the class (instead of just one, as in this example), it is somewhat easier to call them using the name of the class than through a name bound to a single instance of the class. That is because once the name of the class is imported into a module's namespace, it can be used directly, which is much easier than passing an instance among many functions as one of their arguments.

Inheritance

It often happens that you want to design and implement a class that is a lot like another. You could copy the code from the other class to use as a starting point and then modify it slightly, but that strategy has many weaknesses:

- The more copies of a body of code there are, the more work it takes to maintain them.
- The more copies of code there are that need changing, the more likely it is that mistakes will be made in making the changes.
- The larger the code grows, the more there is to read and understand.
- The larger the code grows, the more effort it takes to navigate through and edit the code.

Object-oriented languages provide a powerful mechanism called *inheritance* that allows you to define a new class that differs from an old one only in the methods it defines. This new class is called a *subclass*, and the class on which it is based is called a *superclass*. (The terms *base class* and *derived class* are also sometimes used.) When a class is defined with one or more superclasses, its definition implicitly includes all the methods and class fields of each of its superclasses. Those, in turn, contain all the methods and

class fields of *their* superclasses. Before it defines anything at all, the new subclass inherits all of the methods and class fields of the entire chain of superclasses.

Defining subclasses

A subclass is defined by following its name with the name of its superclass, enclosed in parentheses, as shown in Example 5-5.

Example 5-5. Defining a subclass

```
class BaseSequence:

    def __init__(self, seqstring):
        self.seq = seqstring

    def get_sequence(self):
        return self.seq

    def gc_content(self):
        """"Return the percentage of G and C characters in base_seq"""
        seq = self.get_sequence.upper()
        return (seq.count('G') + seq.count('C')) / len(seq)

class RNASequence(BaseSequence):        # subclass
    # ...
```

> The parentheses can actually contain a list of superclass names. Having more than one superclass is called *multiple inheritance*. Multiple inheritance can be used for a wide range of purposes, from specific implementation techniques to high-level architectural mechanisms. Multiple inheritance is not all that difficult to understand or use in Python, but it leads to complexities and issues beyond the scope of this book.

A method in a class can refer to its superclass by calling the built-in Python function `super`. The word `self` does not appear either before `super` or within its argument list. The call to the function `super` takes the place of `self` in a method call. The result of `super` is an object that represents the same instance as `self`, but for which methods are obtained from the superclass.

> **Python 2:** In Python 2 code you may occasionally see a class defined with `object` as its superclass. In Python 3 `object` is a superclass of every class, so class definitions do not include it explicitly.

We need `super` as soon as we start defining subclasses, because the `__init__` method of a subclass *must* call the `__init__` method of its superclass. This is so that each class can initialize the fields of the new object according to its own `__init__` procedure. Example 5-6 shows a small example.

Example 5-6. A basic subclass

```
class RNASequence(BaseSequence):

    CodonTable = {'UUU': 'F', 'UCU': 'S', 'UAU': 'Y', 'UGU': 'C',
                  #...
                 }

    @classmethod
    def translate_codon(self, codon):
        return self.CodonTable[codon.upper()]

    def __init__(self, seqstring):
        super().__init__(seqstring)     # note that super does not use self

    def translate(self, frame=1):
        """Produce a protein sequence by translating this
        RNA sequence starting at frame 1, 2, or 3"""
        return ''.join([self.translate_codon(self.get_sequence()[n:n+3])
                        for n in
                        range(frame-1,
                              # ignore 1 or 2 bases after last triple
                              len(self.get_sequence()) -
                              (len(self.get_sequence()) - (frame-1)) % 3,
                              3)])
```

This example defines the class BaseSequence, which simply stores a sequence (presumably a string) and provides an access method for it. It also defines RNASequence as a subclass of BaseSequence. Because BaseSequence defines get_sequence, that method can be called on instances of RNASequence. The translate method can be called on an instance of RNASequence, but not on an instance of the more general BaseSequence, since it doesn't define that method. Although an RNASequence has no initialization work to do for its own purposes, it must forward to BaseSequence.__init__ the arguments it expects:

```
>>> rnaseq1 = RNASequence('UUACCUGGCAUUGCAA')
>>> rnaseq.getsequence()
'UUACCUGGCAUUGCAA'
>>> for n in range(1,4):
        print('Frame', n, rna1.translate(n), sep = '  ')
Frame  1  LeuProGlyIleAla
Frame  2  TyrLeuAlaLeuGln
Frame  3  ThrTrpHisCys
```

Initialization functions often check the validity of their arguments. We'll extend the definition of RNASequence.__init__ to use a class method that raises an exception if there is an invalid character in seq:

```
    @classmethod
    def invalid_chars(self, seqstring):
        return {char for char in seq if char not in 'UCAGucag'}

    def __init__(self, seqstring):
        invalid = self.invalid_chars(seqstring)
```

```
        if invalid:
            raise Exception(
                'Sequence contains one or more invalid character]: ' +
                str(invalid))
        super().__init__(seqstring)
```

Next, we'll add a `DNASequence` class with a very slightly different definition of `isinvalid`. Instead of `translate`, it includes a `transcribe` method that returns its RNA complement:

```
class DNASequence(BaseSequence)
    # construct a string translation table for use with str.translate
    TranslationTable = str.maketrans('TCAGtcag', 'AGUCaguc')

    @classmethod
    def invalid_chars(self, seqstring):
        return {char for char in seq if char not in 'TCAGtcag'}

    def __init__(self, seqstring):
        invalid = self.invalid_chars(seqstring)
        if invalid:
            raise Exception(
                'Sequence contains one or more invalid characters: ' +
                str(invalid))
        super().__init__(seqstring)

    def transcribe(self):
        """Produce an instance of RNASequence that is
        the transcribed complement of this sequence"""
        return RNASequence(      # use string translation table to transcribe
            self.get_sequence().translate(self.TranslationTable))
```

Classes related by inheritance form a *class hierarchy*. Conceptually, this is a taxonomy of types, analogous to a biological taxonomy. However, the relationship is more than just conceptual, due to the technical consequences of the superclass/subclass relationship. The (simple) taxonomy for the preceding example is shown in Figure 5-1.

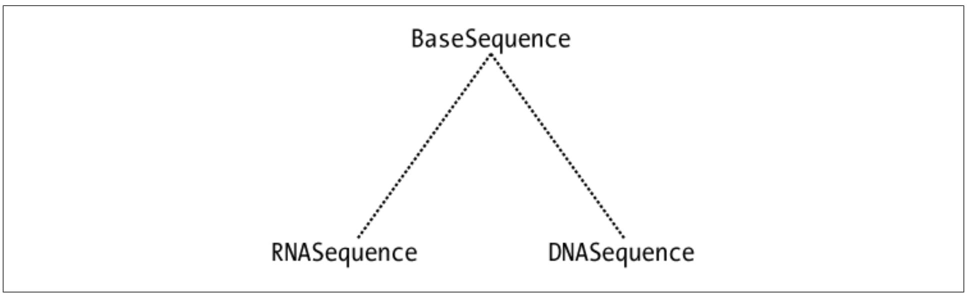


Figure 5-1. A class hierarchy

Factoring out common code

Notice the repetition of code in RNASequence and DNASequence in the steps taken to validate the seq argument of __init__. The only difference is two characters in the string of invalid_chars. Otherwise, invalid_chars and __init__ are exactly the same in the two classes. Were we to notice such duplication in an ordinary function, we would pull out the common code and put it in a new function, calling it from the old ones. Slight differences in the old code would be handled by adding parameters to the new function.

With classes, we can do even better than that: methods that are duplicated in sibling classes can be moved up to a common superclass. Removing such duplication is one of the most common uses of inheritance. By analogy with the similar notion from algebra, this reorganization is often called *factoring out*.

Example 5-7 shows revised—and shorter—definitions of the classes. The method invalid_chars has been moved to BaseSequence, and the test for invalid characters has been moved from RNASequence.__init__ and DNASequence.__init__ into BaseSequence__init__.

Example 5-7. Factoring out common code to a superclass

```
class BaseSequence:

    @classmethod
    def invalid_chars(self, seqstring):
        return {char for char in seq if char not in self.ValidChars}

    def __init__(self, seqstring):
        invalid = self.invalid_chars(seqstring)
        if invalid:
            raise Exception(
                type(self).__name__ +
                'Sequence contains one or more invalid characters: ' +
                str(invalid))
        self.__seq = seqstring

    def get_sequence(self):
        return self.__seq

    def gc_content(self):
        """"Return the percentage of G and C characters in the sequence"""
        seq = self.get_sequence.upper()
        return (seq.count('G') + seq.count('C')) / len(seq)

class RNASequence(BaseSequence):

    ValidChars = 'UCAGucag'
    CodonTable = {'UUU': 'F', 'UCU': 'S', 'UAU': 'Y', 'UGU': 'C',
                  # ...
                  }

    @classmethod
    def translate_codon(self, codon):
```

```
        return self.CodonTable[codon.upper()]

    def __init__(self, seqstring):
        super().__init__(seqstring)

    def translate(self, frame=1):
        """Produce a protein sequence by translating this
        RNA sequence starting at frame 1, 2, or 3"""
        return ''.join([self.translate_codon(self.get_sequence()[n:n+3])
                        for n in
                        range(frame-1,
                              # ignore 1 or 2 bases after last triple
                              len(self.get_sequence()) -
                              (len(self.get_sequence()) - (frame-1)) % 3,
                              3)])

class DNASequence(BaseSequence):
    ValidChars = 'TCAGtcag'
    # construct a string translation table for use with str.translate
    TranslationTable = str.maketrans('TCAGtcag', 'AGUCaguc')

    def __init__(self, seqstring):
        super().__init__(seqstring)                        # note absence of self

    def transcribe(self):
        """Produce an instance of RNASequence that is
        the transcribed complement of this sequence"""
        return RNASequence(
            self.get_sequence().translate(self.TranslationTable))
```

To generalize the definition of `invalid_chars`, the strings the old definitions contained have been replaced by a class variable, `ValidChars`. When the method is executed, `self` will be an instance of either `RNASequence` or `DNASequence`, so in evaluating the expression `self.ValidChars` Python will look first in the instance, then in the instance's class, where it will find the value. (If the instance's class did not define the class field, Python would next look for it in the superclass.) Example 5-8 demonstrates this process in the creation of a new `DNASequence`.

Example 5-8. Referencing a subclass class value

```
DNASequence('ATTCTGGC')
    → DNASequence.__init__('ATTCTGGC')
        → BaseSequence.__init__('ATTCTGGC')
            → BaseSequence.isinvalid('ATTCTGGC')
                  which references self.ValidChars
                       1. does self have a ValidChars field? no
                       2. does self's class have a ValidChars field? yes
                       [3. if it did not, the superclass of self's class
                            would be examined for a ValidChars field]
```

Generalization

It is often the case that after defining two apparently independent classes, some commonality becomes apparent. Such situations motivate the definition of a new class. This new class is a *generalization* of its subclasses. In truth, any class can be described as a generalization of its subclasses, but the term specifically calls attention to the action of creating a new superclass to capture code common to some existing classes.

As an example of generalization, suppose we want to add `ProteinSequence` to our group of classes. It will share some, but not all, of `BaseSequence`'s implementation. Therefore, we'll define a more generalized class called `Sequence`. This leads to the following changes:

- `BaseSequence` is changed to inherit from `Sequence`.
- A new class, `ProteinSequence`, is defined with `Sequence` as its superclass.
- The initialization of `seq` is moved to `Sequence.__init__`.
- The method `sequence` is moved to `Sequence`.
- The method `gc_content` *stays* in `BaseSequence`, because it is only applicable to base sequences, not to amino acid sequences.
- The method `valid_chars` is moved to `Sequence`.

While we're at it, we can also add a new class called `AmbiguousDNASequence`, which allows sequences to have ambiguity codes in addition to DNA symbols. At first sight it might look like this should be a subclass of `DNASequence`. However, ambiguous sequences cannot be transcribed into `RNASequence`s, and it is rarely appropriate to define a subclass when its superclass has methods that are inapplicable to the new class's instances. Thus, we'll make `AmbiguousDNASequence` a direct subclass of `BaseSequence`.

Now that we have a `ProteinSequence` class we can change `RNASequence.translate` to return an instance of that instead of a string, the way `DNASequence` returns an instance of `RNASequence`. We'll also define a `__str__` method for `Sequence` that simply returns the sequence. Because these classes are all instantiated with a single string argument, we can define an appropriate `__repr__` as well.

For both `ProteinSequence` and `AmbiguousDNASequence`, we'll assign appropriate `ValidChars` values and define the usual `__init__` methods. We'll also give `ProteinSequence` a `long_string` method that returns a string with the one-letter codes replaced by the corresponding three-letter ones. We'll use another class field to define a dictionary to implement that and create a list of valid characters by extracting its keys. Note that during the execution of the class definition statement, class fields are just ordinary names. One assignment can use the result of a previous assignment. There is no `self` at that point, because the class is not defined until the `class` statement has completely finished executing.

Example 5-9 shows the definitions of the new classes and the changes to the old ones.

Example 5-9. Generalized Sequence class and its subclasses

```
class Sequence:

    @classmethod
    def invalid_chars(self, seqstring):
        return {char for char in seq if char not in self.ValidChars}

    def __init__(self, seqstring):
        invalid = self.invalid_chars(seqstring)
        if invalid:
            raise Exception(
                type(self).__name__ +
                " contains one or more invalid characters: " +
                str(invalid))
        self.__seq = seqstring

    def get_sequence(self):
        return self.__seq

class BaseSequence(Sequence):

    def __init__(self, seqstring):
        super().__init__(seqstring)

    def gc_content(self):
        """"Return the percentage of G and C characters in the sequence"""
        seq = self.get_sequence.upper()
        return (seq.count('G') + seq.count('C')) / len(seq)

class ProteinSequence(Sequence):

    ThreeLetterCodes = {  # including ambiguity codes B, X, and Y
                'A': 'Ala', 'B': 'Asx', 'C': 'Cys', 'D': 'Asp',
                'E': 'Glu', 'F': 'Phe', 'G': 'Gly', 'H': 'His',
                'I': 'Ile', 'K': 'Lys', 'L': 'Leu', 'M': 'Met',
                'N': 'Asn', 'P': 'Pro', 'Q': 'Gln', 'R': 'Arg',
                'S': 'Ser', 'T': 'Thr', 'V': 'Val', 'W': 'Trp',
                'X': 'Xxx', 'Y': 'Tyr', 'Z': 'Glx'}

    ValidChars = ''.join(ThreeLetterCodes.keys())
    ValidChars += ValidChars.lower()
    TranslationTable = str.maketrans(ThreeLetterCodes)

    def long_str(self):
        return self.get_sequence().translate(self.TranslationTable)
        # or ''.join([ThreeLetterCodes[key]
        #              for key in self.get_sequence()])

    def __init__(self, seqstring):
        super().__init__(seqstring)

class RNASequence(BaseSequence):

    ValidChars = 'ucagUCAG'
```

```
    CodonTable = {'UUU': 'F', 'UCU': 'S', 'UAU': 'Y', 'UGU': 'C',
                  # ...
                  }

    @classmethod
    def translate_codon(self, codon):
        return self.CodonTable[codon.upper()]

    def __init__(self, seqstring):
        super().__init__(seqstring)

    def translate(self, frame=1):
        return ProteinSequence(
            ''.join((self.translate_codon(self.get_sequence()[n:n+3])
                     for n in
                     range(frame-1,
                           len(self.get_sequence()) -
                           ((len(self.get_sequence()) - (frame-1)) % 3),
                           3))))

class DNASequence(BaseSequence):
    ValidChars = 'TCAGtcag'
    # construct a string translation table for use with str.translate
    TranslationTable = str.maketrans('TCAGtcag', 'AGUCaguc')

    def __init__(self, seqstring):
        super().__init__(seqstring)     # note absence of self

    def transcribe(self):
        """Produce an instance of RNASequence that is
        the transcribed complement of this sequence"""
        return RNASequence(
            self.get_sequence().translate(self.TranslationTable))

class AmbiguousDNASequence(DNASequence):

    ValidChars = 'TCAGBDHKMNSRWVY'
    ValidChars += ValidChars.lower()

    def __init__(self, seqstring):
        super().__init__(seqstring)
```

The inheritance relationships among these classes are shown in class hierarchy diagrammed in Figure 5-2.

The changes made in Example 5-9 left `BaseSequence` empty. Let's add a method that applies to all kinds of `BaseSequence`s, but not to `ProteinSequence`s: `complement`. We'll use translation tables to implement it, as we did for `DNASequence.transcribe`. We'll even take the somewhat unusual step of defining complements for ambiguity codes:

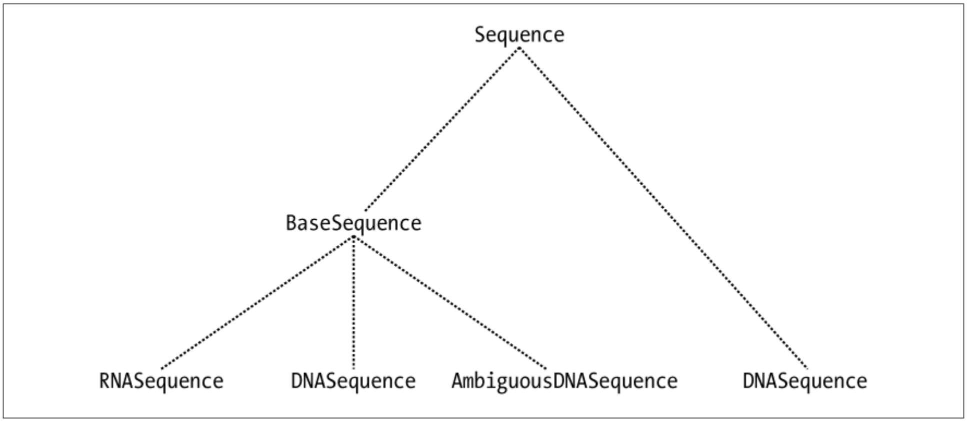


Figure 5-2. A class hierarchy

```
class BaseSequence(Sequence):
    # ...

    def complement(self):
        return self(self.get_sequence().translate(self.ComplementTable))

    def gc_content(self):
        """"Return the percentage of G and C characters in the sequence"""
        seq = self.get_sequence.upper()
        return (seq.count('G') + seq.count('C')) / len(seq)

class RNASequence(BaseSequence):

    ComplementTrans = str.maketrans('UCAGucag', 'AGUCaguc')
    # ...

class DNASequence(BaseSequence):

    ComplementTable = str.maketrans('TCAGtcag', 'AGTCagtc')
    # ...

class AmbiguousDNASequence(DNASequence):

    ComplementTable = str.maketrans('TCAGRYMKSWNBVDH', 'AGTCYRKMSWNVBHD')
    # ...
```

Now we can invoke `complement` on instances of any subclass of `BaseSequence`, getting results appropriate to each type of instance.

Subclass methods

Methods defined in a subclass can serve a number of different purposes relative to the methods of their superclasses:

Addition
: The subclass extends the superclass's repertoire by defining new methods.

Compounding

A subclass method can invoke several superclass methods. This has the effect of combining the actions of those methods.

Blocking

Very occasionally, especially in elaborate inheritance hierarchies, it makes sense to define a class as a subclass of another even if it does not support a few of the superclass's methods. For instance, a simplistic view of real-world biology would say that a penguin *is* a kind of (i.e., subclass of) bird, *except that* it doesn't fly. Usually, though, it is better to *refactor* (i.e., rearrange) the class hierarchy to eliminate this conflict. We did this when adding `ProteinSequence` to our earlier example.

You have three options when defining a method that blocks an inherited one: do nothing, print or log a "not supported" warning, or raise an exception.

Extension

The subclass method calls the superclass method before, in the middle of, or after performing some other action(s). Each class's `__init__` method is necessarily an extension of its superclass's `__init__` method. For other methods, extension can take many forms. The subclass can add additional verification of values used by the superclass's methods, add printing or logging facilities, check security restrictions, store or update information beyond what is managed by the superclass method, and so on.

> An extension method's call to the corresponding superclass method is not limited to passing the values of its parameters. There are many kinds of opportunities for passing values that have been changed or replaced in some way.
>
> Definitions of `__init__` are usually more constrained. They almost always begin with a call to `super.__init__`, with an unmodified set of arguments corresponding to the inherited parameter list. If the subclass's `__init__` defines additional parameters, those will typically be processed after the call to the superclass's `__init__`.

Overriding

There is nothing problematic about defining a method in a subclass that simply replaces one with the same name in its superclass. This is called *overriding*. Unlike extension methods, a method that overrides one its superclass defines does *not* use `super`. Normally, the subclass method serves the same purpose as the superclass method; the only difference is in its implementation.

One method that needs overriding in our example is `gc_content`. In computing the proportion of G and C bases, we divided the total count of the two by the length of the sequence. However, assuming it makes sense to compute `gc_content` for DNA sequences that include ambiguity codes, we need a different definition for `gc_content` in `AmbiguousDNASequence`. In computing the sum of the G and C bases

it should also include occurrences of the ambiguity code S, since that stands for either a G or a C. It should then compare the sum of those counts not to the entire length of the sequence, but rather to the count of symbols that are either definitely G or C (which includes S) or definitely A or T (which includes W). For the purposes of this computation, we must ignore other ambiguity codes. The definition, a typical example of overriding, would be:

```
class AmbiguousDNASequence(DNASequence):

    def gc_content(self):
        """Return the percentage of G and C characters in the sequence, counting
        only bases that are either definitely G or C or definitely A or T"""
        seq = self.get_sequence.upper()
        gc_count = (seq.count('G') + seq.count('C') + seq.count('S')
        at_count = (seq.count('A') + seq.count('T') + seq.count('W')
        return gc_count / (gc_count + at_count)

    # . . .
```

Overriding and extension are different from the other techniques in a very important respect: because the subclass method replaces the superclass method in calls through instances of the subclass, its parameter list should conform to that of the superclass method. That is, it should look like the superclass method so that it can be used without distinguishing the type of the instance through which it is called. This means that the subclass's parameter list should begin with the same parameters as the ones in the superclass method's parameter list.

Other arguments may be added after those for the benefit of code that knows it is dealing with instances of the subclass. Such arguments must be optional so that other code does not need to supply values for them; they must be defined as keyword arguments that follow the parameters of the superclass method.

Tips, Traps, and Tracebacks

Tips

- Don't capitalize the names of class files: Python filenames should be all lowercase. Confusion can result from having modules and classes with the same names. Capitalizing all your class names and leaving filenames in lowercase avoids problems.
- The class definitions in this book's examples generally use the same name for the parameters of `__init__` methods as the fields that get initialized with their values. This is a stylistic preference, not a requirement.
- Most methods of a class should use its own `get` and `set` methods to access its fields instead of accessing them directly. The advantages of information hiding are

important for the class developer too, not just for other programmers who use the class. It may not seem important when you first develop a class, but when you come back to it later, it is easy to forget what the fields contain. Also, many `get` methods may compute, search for, or look up values that aren't directly stored in a field. Dictionary-valued fields are particularly problematic in this regard:

— Quite often, other methods in the class will do something with the dictionary's entries. They should call `get_entries`. Otherwise, when the developer comes back to the class after a while and sees a field with the name `entries`, he may well try to use it directly, thinking it is a sequence. The errors that result when keys are used instead of values are distracting at minimum, and often quite difficult to decipher.

— A very common mistake with dictionary-valued fields is using them directly in an iteration, comprehension, or generator expression without calling `keys`, `values`, or `items`. Doing so is the same as calling `keys`, which is not always what you want. An access function `get_entries` that accesses the values of a dictionary-valued field would normally be coded as a generator expression:

```
def get_entries(self):
    return (entry for entry in self.entries.values())
```

— Calling the field `entry_dict` might avoid that particular confusion, but there is still the problem that if the representation is changed from a list to a dictionary, or the other way around, the name of the field will have to be changed throughout the class. More importantly, the code that used to use `entry_dict.values` will have to be changed to use `entries` directly. This can be avoided by using `get_entries` in the class's own methods. Then, a change in the representation requires changing only `get_entries` (and perhaps one or more "set" methods).

- When extending an inherited method, think carefully about whether you want the superclass's method to be called before, in the middle of, or after the body of the subclass's method. For `__init__`, the call to `super` should almost always be first. Otherwise, it depends on the way the methods are designed to work, especially if the superclass method calls others.

Information Hiding

We have talked extensively about defining a functional interface to classes and instances and insisting that external code use it. Yet the mechanisms and techniques shown in this chapter are insufficient to prevent external code from directly accessing class and instance fields or calling methods meant to be used only from other methods of the class. Information hiding can be enforced in Python at two different levels:

- Prefixing a method or field with a single underscore is a convention indicating that those names should not be used outside the class, except perhaps by closely related classes. It has no effect, but it does highlight the class implementer's intention that the method or field should not be used outside the class.

- Prefixing a name with two underscores makes that name unavailable outside the class; this restriction *is* implemented by Python. For example, if an `__init__` method assigned `self.__name`, external code would not be able to access that field —it would be forced to use the `get_name` method the class presumably supplies.

Do not *end* a name with two underscores—this is a different convention than Python's special methods, which both begin and end with two underscores.

Traps

- Omitting `self` from the parameter list of a method is a common mistake. The error that results is usually a complaint about the method being called with one too many arguments.
- Omitting `self.` before a reference to a field or method of the same class inside of a method is another common mistake; this will typically manifest itself as a name error, since the field or method name is not likely to have a binding outside of the instance's namespace.
- If an instance is lacking a field you expected to find, make sure the assignments in `__init__` begin with `self`. Otherwise, the names assigned will just be local to the method's namespace, and they'll disappear when the method exits.
- No assignments at the top level of a class, including assignment of default values for method parameters, may refer to `self`. The class does not exist until its definition statement has completed execution. At that point Python attaches its namespace to it. The same thing happens with modules: a module being imported does not get created until all of its statements have been executed, at which point Python creates a module object and attaches to it the namespace containing its definitions.
- In an instance method, reassignment to a class field through `self` assigns that field in the instance, not the class field as intended. You must use the name of the class to rebind the class field from within an instance method: for example, `GenBankEntry.count` instead of `this.count`. You can *access* the value of the class field through `self`, but you cannot rebind it that way. Assigning a name through `self` binds a field in that instance the way assigning a name inside a function definition binds that as a name to the function.
- When you run Python from the command line or run a program using IDLE's Run menu command (F5), your classes are loaded into a clean Python. If, however, you are using Python interactively outside of IDLE, you must understand the effects of executing and importing code. This is much more of an issue when using classes, and even more so with multiple modules:
  - Reimporting a module *has no effect*: Python keeps a list of modules that have been imported and doesn't import ones that are on that list. There are ways to "reload" modules, but they are not meant for casual use. If you are not using IDLE (or another IDE that clears the status of the Python environment when a

file executes), and you are executing the code in one file that imports code from another file that has changed, you pretty much have to restart Python.

— Trouble can arise with old definitions even if you have all your code in one file. When you reexecute a class's definition, the old instances continue to refer to the old class definition, with its old method definitions. A module that creates instances of another module's classes may also end up referring to the old classes. This is a very confusing area of Python use. While inconvenient in some respects, IDLE's automatic restart eliminates this problem too. If you are getting mysterious results after editing and reexecuting class definitions outside of IDLE, try restarting Python.

Tracebacks

Here are some representative error messages:

`AttributeError: type object 'super' has no attribute 'draw'`
: `super` was used without parentheses.

`NameError: global name '`*`attribute`*`' is not defined`
: The name *`attribute`* (a field or method) was not preceded by `self.` inside a method.

`TypeError:` *`method`*`() takes no arguments (1 given)`
: A method's parameter list did not include `self`.

`TypeError: unorderable types:` *`Classname`*`() <` *`Classname`*`()`
: No `__lt__` method was defined for *`Classname`*, and `sorted` was called on a sequence of its instances or `list.sort` on a list of its instances.

The following does not cause a traceback, but it is probably not what you expected when you tried to call a method such as `get_gid` while omitting the parentheses:

```
>>> gb
 <GenBankEntry 1293613 U49845.1 'Saccharomyces cerevisiae'> # uses __repr__
>>> gb.get_gid
<bound method GenBankEntry.get_gid of     # line break inserted manually for formatting purposes
 <GenBankEntry 1293613 U49845.1 'Saccharomyces cerevisiae'>>
```

This is simply another form of leaving out the parentheses in an expression intended to be a function call. For instance:

```
>>> print
<built-in function print>
```

A *bound method* is a method associated with (bound to) a particular instance. Before it is bound, it is just a function like any other:

```
>>> GenBankEntry.get_gid
<function get_gid at 0x1c7d228>
```

Once it is called through an instance, however, the method's type changes to bound method and its printout includes the instance to which it is bound.

CHAPTER 6
Utilities

This chapter surveys many Python modules that are particularly important for bioinformatics programming. We will look at utilities that help you connect Python to its external environment, deal with filesystems, work with text, and debug your programs, among others. These aren't particularly exciting, but it's easy enough to learn how to use them. You'll need some of them to deal with operating systems, filesystems, dates and times, and so on. Others will likewise prove extremely useful in the right circumstances, giving you ways to do in a few lines of code things that could take pages otherwise. Some of the modules aren't necessary for simple scripts or initial development but provide the features users expect from mature programs.

System Environment

The first set of modules we'll look at are those that allow Python to interact with its external environment. The categories included here are dates and times, command-line processing of various kinds, email, and logging.

Dates and Times: datetime

The `datetime` module defines classes that represent a date, a time, a combination of date and time, and a couple of others. There are some basic ways of manipulating instances of these classes, as well as some more elaborate ones that we won't consider here.

Classes

The `datetime` module defines five classes. Their representations are simple, consisting only of several fields with primitive types. This allows them to define `__repr__` methods that print exactly what you would enter to create one. You'll see such output when you enter into the interpreter an expression whose value is an instance of one of these classes. They also define `__str__` methods for the more human-oriented output used by `print`. The five classes are:

`datetime.date`
: Represents a date with attributes `year`, `month`, and `day`

`datetime.time`
: Represents a time with attributes `hour`, `minute`, `second`, `microsecond`, and `tzinfo` (time zone information)

`datetime.datetime`
: Represents a combination of `date` and `time` with the attributes of each

`datetime.timedelta`
: Represents the difference between two `date`s, two `time`s, or two `datetime`s, with attributes `days`, `seconds`, and `microseconds`

`datetime.tzinfo`
: Represents time zone information

Note that unlike many classes, these don't define "access" methods to get the values of their attributes. Instead the attributes are accessed directly using dot notation, e.g., `date1.year`. It is very important to note that as with strings, instances of the classes in this module are immutable. There are methods to create one instance based on the values of another's attributes, but once an instance is created its attribute values cannot be changed.

Instance creation

Like any other type, the types in the `datetime` module can be called as functions to create new instances:

```
datetime.date(year, month, day)
datetime.time(hour[, minute[, second[, microsecond[, tzinfo]]]])
datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[,
tzinfo]]]]])
datetime.timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[,
hours[, weeks]]]]]]])
```

The classes also provide methods to create instances based on the current date and/or time:

`datetime.date.today()`
`datetime.datetime.now([tz])`
`datetime.datetime.combine(date, time)`
: Return a `datetime` with the date portion taken from *`date`* and the time portion taken from *`time`*

There are also methods that create new instances from existing ones, possibly of a different class:

date1.replace(*year*, *month*, *day*)
: Returns a new date with the same attribute values as date1, except for those for which new values are specified (usually as keyword arguments)

time1.replace([*hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])
: Returns a new time with the same attribute values as time1, except for those given new values by whichever keyword arguments are specified

datetime1.date()
: Returns a new date with the day, month, and year of *datetime1*

datetime1.time()
: Returns a new time with the hour, minute, second, and microsecond of datetime1

datetime1.replace([*year*[, *month*[, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]]]]])
: Returns a new datetime with the same attribute values as *datetime1*, except for those given new values by whichever keyword arguments are specified

Operations

Instances of these classes support a few arithmetic and comparison operations. Table 6-1 shows the arithmetic operations supported by instances of date.

Table 6-1. Operations supported by date instances

Operation	Result
date1 + *timedelta1*	*date1* + *timedelta1*.days
date1 - *timedelta1*	*date1* - *timedelta1*.days
date1 - *date2*	A timedelta
date1 < *date2*	True if *date1* is earlier than *date2*

The same operations are supported by instances of datetime with equivalent interpretations. However, the only operation supported by time instances is <. Table 6-2 shows operations that can be performed with instances of timedelta.

Table 6-2. Operations supported by timedelta instances

Operation	Result
t1 + *t2*	Sum
t1 - *t2*	Difference
t1 * *i*, *i* * *t1*	Product of *t1* and integer *i*
t1 // *i*	"Floor" division, as with numbers
-*t1*	A new timedelta whose attribute values are the negation of *t1*'s

Operation	Result
`abs(t1)`	A copy of *t1*, except that a negative value of `days` is made positive
`comparisons`	The usual six: `==`, `!=`, `<`, `<=`, `>`, `>=`

Methods

In addition to being able to access the attributes described in the preceding sections using the usual dot notation, the following methods are often useful (the module's classes define many more, but most are used primarily in low-level system development):

date1`.isoweekday()`
: Returns the day of the week of ***date1*** as an integer, with Monday = `1`

date1`.isocalendar()`
: Returns a tuple of ***date1***'s values with the following elements:

    (*ISO year, ISO week number, ISO weekday*)

date1`.isoformat([`*sep*`])`
: Returns a string representing ***date1*** in the standard format:

    YYYY-MM-DD

time1`.isoformat([`*sep*`])`
: Returns a string representing ***time1*** in the standard format:

    HH:*MM*:*SS*.*mmmmmm*

    omitting the microseconds if `0`

datetime1`.isoweekday()`
: Returns the day of the week of ***datetime1*** as an integer, with Monday = `1`

datetime1`.isocalendar()`
: Returns a tuple of `datetime1`'s values with the following elements:

    (*ISO year, ISO week number, ISO weekday*)

datetime1`.isoformat([`*sepchar*`])`
: Returns a string representing ***datetime1*** in the standard format:

    YYYY-MM-DDsepcharHH:*MM*:*SS*.*mmmmmm*

    omitting the microseconds if `0`; ***sepchar*** is an optional one-character string that separates the date and time portions of the string (`'T'`, if omitted)

System Information

When a program is started, it may be given command-line arguments that are in effect parameters of the program itself. While it is executing, it can read from and write to

predefined standard streams that connect it to the environment in which it is running. These fundamental features, along with access to details about Python's internal environment, are implemented by the `sys` module. Programs may also occasionally need to access aspects of the external environment, including the computer's internal clock and information about other time-related details such as the time zone and daylight savings time settings. The `time` module provides facilities for that.

The Python runtime environment: sys

The `sys` module provides access to a great deal of technical information about the Python implementation. It provides mechanisms for modifying various aspects of the interpreter, such as the prompt. Following are frequently used values defined by the module (remember, you must use either `import sys` or `from sys import` *`name`* for whatever *`name`* you want to use):

`sys.argv`
: This is the single most important name defined in `sys`. When Python is run from the command line, `sys.argv` is a list of strings containing the parts of the command line. The first—`sys.argv[0]`—is the name of the Python program being run. You will use this often. Consider the following very brief demonstration program:

```
# File showargs.py
import sys
for arg in sys.argv:
    print(arg, end=' ')
```

Running this program from the command line simply prints the name of the file and the command-line arguments:

```
% python showargs.py a 1
showargs.py a 1
```

If your program expects a certain number of arguments, you should always check that no fewer arguments than expected and no more than allowed have been provided. This should be done at the end of the file, inside a test that determines whether `__name__` is `'__main__'`, since you wouldn't check for command-line arguments if the module were imported. (Remember that `sys.argv` always includes the name of the Python file being executed, so its length is always at least `1`.) A Unix-style "usage" message should be printed when an unacceptable number of arguments are provided, and the program should exit. A typical construction is:

```
if __name__ == '__main__':
    if len(sys.argv) < 2:
        print('Usage: downloadPDFs filename ...')
    else:
        for filename in sys.argv[1:]:
            process(filename)
```

`sys.modules`
: This is a dictionary whose keys are the names of the currently loaded modules and whose values are the corresponding module objects. These are the modules that have been loaded into the system. When Python first starts there are 50–75 modules in the system, depending on how it was compiled; the version; and the platform, among other details. Most of the preloaded modules are low-level implementation facilities, but some of them are modules we'll be discussing and you'll be importing. Although these modules are already in Python, their names are not in the interpreter's namespace until you import them. Modules imported by your code get added to this list.

  This value is mentioned only as a view into a bit of Python's implementation. You won't do anything with this value, but you can look at it if you're curious. Most importantly, it shows how simple and consistent Python's implementation is: even its collection of modules is an ordinary dictionary. In fact, module objects are themselves very little more than dictionaries.

`sys.builtin_module_names`
: This is another value that you might find interesting but will rarely need to look at, and certainly will never need to change. It's simply a list of the names of the modules from which the system was built. Most, but not all, are names of modules that are also imported and are therefore in the `sys.modules` dictionary.

`sys.path`
: This is a list of directories where Python looks for modules when it executes an `import` statement. By customizing this value, you can inform Python about directories containing Python modules you use in your code (these could be libraries you've downloaded, libraries your organization has built, or libraries of code you've developed yourself).

  When IDLE is started, a platform-dependent default directory is added to the front of `sys.path`—for example, on OS X it would be the user's *Documents* directory. When the interactive interpreter is started from the command line, the command line's "current directory" is added to the front of `sys.path`. When a Python file is executed, either from the command line or from within IDLE, the file's directory is added to the front of `sys.path`.

`sys.stdin`, `sys.stdout`, `sys.stderr`
: Languages whose programs can be run from a command line generally use three default streams for their input and output, called `stdin`, `stdout`, and `stderr`. As their names suggest, the first is for input, the second is for output, and the third is for error messages. The `sys` module defines names for these three streams.

  The `input` function always reads from `sys.stdin`. However, the input stream is not necessarily taken from the user's typing—a file or the output of another process could be redirected to your program, which could read it with `input`. You could also call `sys.stdin.readline` or use any of `file`'s other input methods.

The significance of having separate streams for output and error messages is that one or the other can be "redirected" to a file or another program, and it's important to be able to separate normal from error output. In particular, you will often redirect standard output to a file but leave error output printing to the command-line environment, where you can see it immediately. You can use either `sys.stdout` or `sys.stderr` as the optional `file` argument to `print`, or call any of `file`'s output methods on them.

A subtle difference between `sys.stdout` and `sys.stderr` is that `sys.stdout` is *buffered* while `sys.stderr` is not. Low-level operating system input and output operations involve significant overhead. Instead of incurring that overhead for every character or line, output to `sys.stdout` is accumulated and written only in reasonable-sized chunks. Error messages, however, usually need to be seen immediately, so each line printed to `sys.stderr` is delivered immediately.

`sys.exit([`*`arg`*`])`
: A function definition can exit a program by calling `sys.exit`, which returns ***`arg`*** (default `0`) to the operating system. (The Unix convention is that `0` signifies no error, `1` signals a runtime error, and `2` indicates a command-line error.)

When a function is defined with a parameter for the stream (open file) the function should print to, that parameter appears in three places:

- In the function definition
- In every call to the function
- In every call to `print` or any other output function within the definition

Some programs contain many functions that write output to the same destination. It becomes very tedious to have to specify that destination in every function definition, function call, and call to `print`. An alternative approach is to omit the destination parameter from the function definitions, function calls, and `print` calls, and let all the output default to `sys.stdout`. To write the output to the desired file, the program can simply rebind `sys.stdout` to the result of a call to `open`. (Except for keywords, there are no sacred names in Python—any name can be rebound with an assignment statement.) If the program will need the original value of `sys.stdout`, it can save the old value, assign the new value, and, in a `finally` clause of a `try` statement, rebind it to its original value. This is shown in the following template.

TEMPLATE

Redirecting to stdout in a Program

You can simplify a portion of a program with a large number of function definitions and calls that all write output to the same destination by rebinding `sys.stdout`. This is a lot easier than including that output destination as a parameter to every function, an argument in every function call, and a `file` keyword argument in every call to `print`.

```
stdout_save = sys.stdout
try:
    sys.stdout = open(filename, 'w')
    ... statements ...
finally:
    sys.stdout = stdout_save
```

The system clock: time

The `time` module does not define any classes. It provides functions for accessing the system's clock, converting among time formats, and working with details of times such as time zones and daylight savings time settings. Most of them are very technical, so we won't discuss them here. A few of the functions, though, are important.

> Computer time is defined relative to a zero starting point sometimes called the *epoch*. Different operating systems define the starting point differently. The absolute value is almost never relevant—what is usually significant is the difference between a clock value saved at one point in the program and the clock value sometime later in its execution.

`time` functions you are likely to find useful include:

`time.gmtime(0)`
: Returns a data structure that describes time zero (in case you are curious)

`time.ctime([`*`seconds`*`])`
: Returns a string showing the date and time at ***`seconds`*** past time zero, defaulting to `nowtime.time`

`time.clock()`
: Returns the number of seconds past time zero, expressed as a floating-point number—use `round` to convert this value to an `int`

`time.sleep(`*`seconds`*`)`
: Stops executing for ***`seconds`*** amount of time, then resumes

The `clock` function is useful for timing the execution of Python programs when you are trying to figure out how to make them run faster. The `sleep` function is used mostly when the program must wait for some external condition. The next template shows how that works.

TEMPLATE

Waiting on an External Condition

Many programs contain a function that must wait until some external condition has been satisfied. A typical example is a file becoming available for reading.

```
while not external-condition-test:
    optional-waiting-action
    time.sleep(seconds)
```

The optional waiting action in the template might be something like printing a period (`.`) to the terminal to show the user that the program is waiting. Without the delay built into the loop, the program would repeat the test and action immediately over and over again, wasting processing resources that the computer could better use doing something else.

Command-Line Utilities

Next, we'll look at utilities for processing information provided by the command-line invocation of a program. We'll also see some utilities that go in the other direction, invoking command-line commands.

Reading multiple files: fileinput

Many programs you write will take a list of files as their command-line arguments and do the same thing with each of them. While you could write a loop that opens and processes each one, sometimes you'll want to treat the contents of the files as one continuous stream of lines. You may also want information about how many lines have been encountered, which file is being read, and so on. The `fileinput` module is a convenient tool to use in these situations in place of writing your own code.

The simplest use is:

```
import fileinput
for line in fileinput.input():
    process(line)
```

By default, `fileinput.input` uses `sys.argv[1:]` as the list of files to iterate over. However, you can give it a different list as an argument. An unusual feature of this module is that it creates a global state that can be accessed through its functions—you don't even need to create and manage an instance of a class (it does that for you). The state functions include:

`fileinput.filename()`
: Returns the name of the current file

`fileinput.filelineno()`
: Returns the number of the line just read from of the current file

`fileinput.isfirstline()`
: Indicates whether the line just read is the first of its file

`fileinput.lineno()`
: Returns the cumulative line number of the line that has just been read

`fileinput.nextfile()`
: Closes the current file, skipping the rest of its lines

Command-line options: optparse

You will sometimes want your Python code to handle the kinds of command-line options you've seen in programs you've used. Dealing with various option formats and analyzing all of a program's command-line arguments to see which are options, which options they are, and what values (if any) they have can turn into a mind-boggling exercise. Even if you write some simple code to handle a couple of options, you may well add more options in the future, and the code will grow in complexity.

The `optparse` module does all the command-line argument parsing for you. You just have to tell it what options the program can take and a bit of information about each. When it's done its work, you are left with an object containing values for the options and a separate list of the arguments that follow the options. Also, `optparse` can generate `-h` and `--help` options automatically from the options you describe.

The `optparse` module implements a large set of features. Just a few of the most important are described here.

TEMPLATE

Using optparse

The basic use of the module follows the outline:

```
import optparse
optparser = optparse.OptionParser(usage=usage_string)
optparser.add_option(details)
optparser.add_option(details)
# ...more options...
optparser.set_defaults(option default values as keyword arguments)
options, args = optparser.parse_args()
```

The first value returned by `parse_args` is a dictionary-like object whose values can be accessed using keyword notation (i.e., `options.`*`key`*). The other value is a list of the positional arguments that followed all the options on the command line.

The optional usage string is normally in the typical Unix format. For example, a program to copy something from a source to a destination might have a usage string such as the following:

```
'usage: copy [options] source destination'
```

If the program can take an indefinite number of arguments, you would end the string with an ellipsis:

```
'usage: copy [options] filename...'
```

When the program prints the usage string, `%prog` will be replaced with the name of the program from the command line (`sys.argv[0]`). (Sometimes it's useful to use filesystem

links so you can call the same program with different names and do something different depending on which name is used; that's a reason for using %prog other than just for convenience.) If you don't provide a usage string, a default value will be used:

```
"usage: %prog [options]"
```

The details part of each call to add_option provides two kinds of information:

- One or more strings indicating the way the option can be specified on the command line; for example:

  ```
  parser.add_option('-o', '--output', '--output_filename')
  ```

- Optional keyword arguments ("attributes") giving further details about the option

Typically, two strings are provided, following the usual Unix style: a "short" name beginning with a single dash, and a "long" name beginning with two dashes. The value of the option specified on the command line (if any) is called the *option argument*. The arguments of both short and long option names may be separated from the names by whitespace. Arguments of short option names may also appear directly following the option name with no intervening whitespace. A long option name's arguments may be separated from the name by an equals sign, without whitespace.

No other syntax for option arguments is supported. Therefore, the possible ways to provide an option argument on the command line are:

```
-o value
-ovalue
--option value
--option=value
```

The basic keyword arguments are shown in Table 6-3, along with their default values.

Table 6-3. Option parser attributes

Name	Default	Description
action	'store'	What the option does (see below)
type	'string'	Type of argument expected (see below)
nargs	1	For *n*, the value of the attribute; if *n* > 1, the option takes the next *n* values from the command line and returns them in a tuple
default	None	The default value for the option
choices		A list of strings that are the valid values for the option
dest	Name of option	The name to use for the option in the option object returned by parse_args
help		Help text for the option

An option's help text should be a string with no line feeds. When help is printed, the module formats the help strings appropriately.

The option's destination is the key that will be used in storing the option's value in the option dictionary. If an option's specification does not include dest, its destination is

determined by its names. If there are one or more "long option" names—those beginning with two dashes—the first is used as the destination. If there is no such name, the first "short option" name—which begins with one dash—is used. Default destinations do not include the initial dashes of the option name.

Option default values may be set in one of two ways: you can include default values in the option specifications, or you can call the parser's `set_defaults` method. That method takes a series of keyword arguments. Each keyword is the destination name of an option, and its value is the default to use if the user of the program does not specify a value for that option on the command line. The method must be called before the option specifications are parsed.

Table 6-4 summarizes how each option type is parsed.

Table 6-4. Option parser types

Types	How parsed
`string`	As is
`int`	`0x` for hexadecimal, `int` if no prefix
`float`	`float()`
`choice`	Like `string`; check by calling `check_choice()`

Table 6-5 describes option parser actions.

Table 6-5. Option parser actions

Action	Consequence
`store`	The option must be followed by an argument on the command line; the argument will be converted according to the option's `type` and stored in the option object under the name give by its `dest`
`store_true`	No command-line value; store the value `True`
`store_false`	No command-line value; store the value `False`
`append`	Like `store`, but the command line can contain multiple occurrences of the option and the `dest` value will be a tuple

The value of the `-h`/`--help` option is generated automatically. The help message will show the usage string (if provided) and the name of each option followed by the value of its `help` attribute. When a help option is specified on the command line, the program exits after showing the help text. Your program can also call the `print_help` function on the instance of `OptionParser`, which prints the help text but does not exit the program. If `%default` appears in an option's help string, the value of the option's default will appear in its place when the help message is printed. Examples 9-4 and 9-5 show the use of `optparse` in a program to download the links contained in a web page (see "Downloading a Web Page's Linked Files" on page 334).

Command-line commands: subprocess

Sometimes you need a program to execute a command as if it had been typed at the command line. This can be a very powerful way of driving other programs from a Python script, controlling the sequence in which they are called, and coordinating their results. It can also be used to invoke command-line facilities rather than programming their equivalents when they don't exist in Python.

The `subprocess` module provides sophisticated, highly generalized facilities that can be used in various ways. The following two high-level functions provide access to most of the module's capabilities (in both, ***`command`*** can be either a string or a sequence of strings):

`subprocess.getoutput(`*`command`*`)`
: Executes ***`command`*** in a shell and returns its output

`subprocess.call(`*`command, ...keyword args...`*`)`
: Executes ***`command`***, waits until it returns, then returns the "success value" of the command

> The success value is platform-dependent, but it's usually `0`. To fit this into the usual Python-style tests, you should think of the result as indicating whether an error has occurred: `0` means no/false (no error has occurred).

Many optional arguments are available for `subprocess.call`, but we'll consider only the two most important. Example 6-1, presented momentarily, shows the use of this function to approximate the behavior of the Unix `ls` command. A more extensive example follows that one.

Probably the most important keyword argument to consider controls whether ***`command`*** is invoked directly—as, for example, when it is simply the name of a program to run—or is executed by a command shell. The difference is important both for the form of ***`command`*** and for how it is interpreted when executed. The default is to not execute in a shell; specify the keyword argument `shell=True` to change that. Here are the key points to keep in mind:

Interpretation of the `command` *argument*
: If executing directly, ***`command`*** should be either a string containing just the program name or a sequence whose first element is the program name (the rest being the arguments for that program).

  If executing in a shell, ***`command`*** can be specified either as a string or as a sequence. A string will be passed directly to the shell. With a sequence, the first element names the command to execute and the rest of the elements are the arguments to that command.

Contents of the ***command*** *argument*

If executing in a shell, ***command*** can be anything you can type in a shell. It can include wildcard filenames, environment variables (whose names are preceded by a dollar sign), actions that are commands to the shell itself rather than programs to run, multiple commands separated by semicolons, and so on.

If executing directly, no variables will be expanded or wildcard filenames matched.

The other important keyword argument is `cwd`, the value of which is a string representing a path to a directory. If a `cwd` argument is provided, the command (whether in a shell or not) will execute in that directory. Many other keyword arguments are also available, but they are for much more specialized technical uses.

Now, here's the promised approximation of the `ls` command (Example 6-1).

Example 6-1. An approximation of the Unix ls command

```
def ls(path ='.', args = ''):
    """Invoke the shell ls command with args on path"""
    subprocess.call('ls' + ' ' +  args + ' ' + path), shell=True)
```

This definition accepts only one path, but it could easily be changed to accept either one path or a sequence of paths. The important aspect of this definition is the `shell=True` argument to `subprocess.call`. Without that there would be no expansion of wildcards, environment variables, and, in Unix, tildes.

Sometimes it's much easier to use a preexisting command-line command than it would be to write your own program to perform a similar function. Unix provides a huge set of programs to manipulate files and their contents. These are particularly useful when dealing with very large files where reading all the data into Python would be impractical or when you need the efficiency and reliability of built-in programs rather than a Python solution you've written yourself. Also, except for the purposes of experimentation and learning, it's almost never a good idea to code functionality you could find somewhere else—keep in mind that this applies to command-line tools.

Example 6-2 uses the Unix `sort` command to sort the data from a Rebase file by the second value on each line. (In the "bionet" format that value is the sequence recognized by the enzyme.)

Example 6-2. Using the Unix sort command to sort a file

```
"""Produce a file of enzyme names and cut sites from the Rebase bionet file with
prototypes omitted and ^s removed, sorted by recognition site sequences"""

def sort_bionet_file(filename = 'data/Rebase/link_bionet.txt'):
    cleanfilename = write_clean(filename)
    sortedfilename = cleanfilename[:cleanfilename.rfind('.')] + '.sorted'
    # Construct Unix sort command
    command = ('sort -f -k 2 -b -o ' + sortedfilename + ' ' + cleanfilename)
    subprocess.call(command)         # tell Unix to execute the command
```

```
def write_clean(infilename):
    """Write a copy of file named infilename with prototypes and ^s omitted"""
    extpos = infilename.rfind('.')
    outfilename = ((infilename if extpos < 0 else infilename[:extpos]) +
                   '.clean')
    with open(infilename) as infile, open(outfilename, 'w') as outfile:
            write_clean_lines(infile, outfile)
    return outfilename

def clean_line(line):
    """Return line without its prototype (if any) and ^ (if any)"""
    line = line.replace('^', '')
    lpos = line.find('(')
    if lpos >= 0:
        rpos = line.find(')')
        return line[:lpos] + ' '*(rpos - lpos + 1) + line[rpos+1:]
    else:
        return line

def write_clean_lines(infile, outfile):
    for n in range(11):
        line = infile.readline()             # skip first 10 lines (introduction)
    while len(line) > 1:                     # stop at empty line or end of file
        outfile.write(clean_line(line))
        line = infile.readline()
```

The arguments to the `sort` command the example constructs have the following meanings:

`-f`
: Ignore case

`-k 2`
: Sort by the second "field"

`-b`
: Ignore leading blanks in the field; the definition of a "field" (unless a separator character is specified with `-t`) is the transition from nonwhitespace to whitespace, so without `b`, the lines will be sorted by groups according to increasing length of the enzyme name

`-o`
: The name of the output file

Communications

When an application is used seriously it becomes important to keep track of its status and activity. Python provides a range of modules that can be used for this purpose. These include a very elaborate `email` module, which we will not talk about; a much simpler way to send email, which we will talk about; and a feature-rich logging facility.

Sending email: smtplib

The straightforward way to send an email message is to use the *Simple Mail Transport Protocol* (SMTP), which dates from the early 1980s, as implemented by Python's `smtp` module. This protocol is still used by many email programs. The steps involved are creating an instance of `smtplib.SMTP`, creating a connection to an email server, providing a username and password if necessary, then sending a message that begins with text in a specific format:

```
To: username, username, ...
From: sender
Subject: optional subject line
Content-Type: text/plain; charset="us-ascii"

Body of message
```

Although the sender and receiver(s) are included in the message text, they must be submitted separately when the message is sent. For most mail servers, especially ones external to your organization, a username and password are required. If your organization's email server can accept SMTP requests without a username and password, they aren't necessary; otherwise, you can set up an account and password for your program to use.

Example 6-3 shows how to put all of this together, including catching the exceptions that typically occur when something goes wrong. While it could be used for many other purposes, the reason we include it here is that it is a good way for a program to email someone when something needs attention, or even just to say that everything's OK.

Example 6-3. Sending email

```
import smtplib
import socket                    # just for socket.err

def sendmsg(fromaddr, toaddr,
            username=None, password=None,
            subject='', msg='',
            hostname='localhost', port=25):

    # Destination can be a single username or a sequence of them
    dest = toaddr if type(toaddr) == str else ', '.join(toaddr)

    msg = '''To: {1}
From: {0}
Subject: {2}
Content-Type: text/plain; charset="us-ascii"

'''.format(fromaddr, dest, subject) + msg

    connection = None
    try:
        connection = smtplib.SMTP()
        connection.connect(hostname, port)
        if username:
```

```
            connection.login(username, password)
        connection.sendmail(fromaddr, dest, msg)
        print('\nMessage sent from {}:\nto: {}'.
              format(fromaddr, dest),
              file=sys.stderr)
    except smtplib.SMTPConnectError as err:
        print('Attempted conneection to {} on port {} failed',
              format(hostname, port),
              file=sys.stderr)
    except smtplib.SMTPAuthenticationError as err:
        print('Authentication for user', username, 'failed',
              'Invalid username-password combination?',
              sep='\n',
              file=sys.stderr)
    except socket.error as err:
        print('Socket error:', err, file=sys.stderr)
    finally:
        if connection:
            try:
                connection.quit()
            except smtplib.SMTPServerDisconnected:
                pass            # can't quit an unconnected SMTP
```

Logging: logging

The `logging` module provides a wide range of options and facilities. The primary advantage of using a logging facility rather than just writing to a file is that different programs can log to the same file without interfering with each other. Other logging features allow various kinds of control over what is logged and where the output goes.

A fundamental concept of logging is that of a *priority level*. The `logging` module uses the levels *debug*, *info*, *warning*, *error*, and *critical*. They are defined both as functions and, with all uppercase names, values (e.g., `logging.WARNING`). Example 6-4 shows a minimal use of the module. We won't go into its many other options and capabilities here, but you should keep the many possibilities in mind when the time comes that you need a logging facility.

Example 6-4. A minimal logging example

```
import datetime
import logging

LOG_FILENAME = 'temp/logging.out'

# This directs the logging facility to use the named file and to
# ignore any requests for log entries with a level less than DEBUG
logging.basicConfig(filename=LOG_FILENAME,level=logging.INFO)

# This appends an entry to the logfile
logging.info("This message was logged at {}."
             .format(datetime.datetime.now().strftime('%H:%M')))
```

If you examine the contents of the logfile you will notice that each line begins with the log level and the word “root”. In more sophisticated uses of the logging facilities, you can set up hierarchies of logs; the name of the log that actually handles the request will then appear in place of “root”.

The Filesystem

In this section we’ll examine important modules for working with the filesystem.

Operating System Interface: os

The `os` module contains a few environmental values and operations similar to those in `sys`. Mostly, though, it provides operations for manipulating files and directories. It implements many low-level operations that you are unlikely to use, but it also contains some that you will use frequently.

The functions in `os` raise an `OSError` when given arguments of the wrong type, paths that aren’t valid (e.g., a file that doesn’t exist), or paths to files or directories that your program doesn’t have the necessary privileges to read.

Environment access

Before moving on to the main content of the `os` module, it is worth mentioning two variables it provides:

`os.sep`
: The string used to separate path components—`'/'` on Unix-based systems, `'\\'` on Windows*

`os.environ`
: A dictionary containing the names and values of “environment variables” obtained from the operating system environment from which you started Python; assigning `os.environ[`*`varname`*`]` will change the value of the environment variable ***`varname`*** outside, as well as inside, Python

You can obtain the value of an environment variable either by accessing `os.environ` as a dictionary or by using the following function:

`os.getenv(`*`varname`*`[, `*`defaultvalue`*`])`
: A function that returns the value of the environment variable ***`varname`***— i.e., the value of the key ***`varname`*** in the dictionary `os.environ`; if that key is not in the dictionary the function returns ***`defaultvalue`*** if specified, and otherwise `None`

* Remember that backslashes in strings signify special characters, so including a backslash in a string requires two of them; `'\\'` is a “special character” like `'\t'` and `'\n'`.

Managing files and directories

`os` provides various functions that mimic the kinds of file and directory management commands available from the operating system command line. The module contains many other functions like these for more technical purposes, but the following sections outline the ones you're most likely to use.

Directories. Common directory-related functions include:

`os.getcwd()`
: Returns the current "working directory" (like the command-line `pwd`)

`os.chdir(`*`path`*`)`
: Makes *`path`* the working directory

`os.mkdir(`*`path`*`)`
: Creates a directory at *`path`*

`os.mkdirs(`*`path`*`)`
: Creates all the directories along *`path`*

`os.rmdir(`*`path`*`)`
: Removes the directory at *`path`*

`os.removedirs(`*`path`*`)`
: Removes all the directories along *`path`*

Files. Useful file-manipulation functions include the following:

`os.remove(`*`path`*`)`
: Removes the file specified by *`path`*

`os.rename(`*`sourcepath, destpath`*`)`
: Renames the file at *`sourcepath`* to *`destpath`*

`os.startfile(`*`path`*`)`
: Opens the file at *`path`* using its associated application

Directory contents. Two very useful functions in `os` deal with the contents of directories:

`os.listdir(`*`path`*`)`
: This function simply returns a list of the names of the files and directories in the directory specified by its argument. The argument is not optional, though it would be nice if it defaulted to the current directory—i.e., `'.'`. (That said, it's easy enough to define your own version of `listdir` that calls `os.listdir('.')`.) Note that the list is ordered arbitrarily; you will often want to sort it.

`os.walk(`*`path`*`)`
: This is almost a facility on its own. Its purpose is to produce the names of all the files and directories at *`path`* and below. For each directory encountered starting at *`path`*, it produces a tuple of the form:

```
(directory-path, subdirectory-names, filenames)
```

The ***directory-path*** in the tuple is a string indicating the path to the directory starting at the initial *path* argument. The two lists contain directory names and filenames only, not full paths; if you need paths, you simply add the ***directory-path*** to the directory name or filename. The function has several optional parameters, but we'll just deal with the basics here.

The value returned by `os.walk` is another kind of iterable. This special kind of object is integrated with `os.walk` in an interesting way: each time around an iteration over the result of `os.walk` you can remove elements from the ***subdirectory-names*** list, and the walk will skip those as it descends into the directory hierarchy. This is useful for ignoring directories such as those beginning with a period or those containing some kind of configuration information that you don't want included in the walk. (The special paths `'.'` and `'..'`—current directory and parent directory, respectively—are never included in the walk.)

Example 6-5 demonstrates a simple of use of `os.walk`. It prints the names of all the directories and filenames in an initial path and below, ignoring any directories whose names begin with a period. The function `show_in_path` obtains the information to be printed and `show_directory_contents` prints it. Indentation is added by `show_in_path` to indicate the "depth" of each file and directory name. (Depth—the number of parent directories up to the original path—is easily calculated by simply counting the number of separator characters in the path.) Using indentation to indicate depth is a common nongraphical technique for showing tree-structured information, such as classical biological taxonomies and filesystem folders. Filesystems are inherently tree-structured because each directory can contain subdirectories.

Example 6-5. Showing a file tree

```
def show_directory_contents(dirpath, filenames, level):
    print('    '*level, dirpath, sep='')
    for name in filenames:
        print('    '*(level+1), name, sep='')

def show_in_path(startpath, ignoredots=True):
    print(startpath)
    for path, dirnames, filenames in os.walk(startpath):
        for dirname in dirnames:
            if dirname[0] == '.':
                dirnames.remove(dirname)
        show_directory_contents(path[len(startpath)+1:],       # strip dirpath
                                filenames,
                                path.count(os.sep))
```

You might recognize this as a Filtered Do iteration (see "Filter" on page 122 in Chapter 4). It's actually a filtered recursive do, but `os.walk` is handling the recursion. We could instead write this example using the template shown in the previous chapter, where recursive iterations were discussed, but this is only a simple example; for more complex tasks you'll find `os.walk` more convenient than writing equivalent code yourself.

Temporary files: tempfile

Creating a new file for output with a name that doesn't conflict with existing names is something programs sometimes do. You could write your own code, generating a numbered filename, checking whether one with that name exists, and incrementing the number until you find a name for which no file exists. However, there are many problems with that naive approach. The module `tempfile` provides a solid facility that takes care of all the intricacies of creating temporary files. It provides the following functions, among some others that are useful for more technical applications:

`tempfile.makestemp([`*`suffix`*`, [`*`prefix`*`, [`*`dir`*`, [`*`text`*`]]]])`
: Creates and returns the absolute path of a temporary file with a name that includes *`suffix`* at the end (default `''`) and *`prefix`* at the beginning (default `'tmp'`), located in the specified directory (default as described for `tempfile.gettempdir` below), and in binary mode unless *`text`* is given as `True`. (The "s" in the name stands for "secure.")

`tempfile.makedtemp([`*`suffix`*`, [`*`prefix`*`, [`*`dir`*`]]])`
: Creates and returns the absolute path of a temporary directory; *`suffix`*, *`prefix`*, and *`dir`* have the same meaning as they do in `tempfile.makestemp`.

`tempfile.gettempdir()`
: Returns the path of the current default directory for creating temporary files and directories. This may be set by assigning `tempfile.tempdir`. If that is unbound or `None`, the environment variables `TMPDIR`, `TEMP`, and `TMP` are examined, with the value of the first one that has a value being used. If none of those environment variables has a value, on Windows the function will look for the directories *C:\TEMP*, *C:\TMP*, *\TEMP*, and *\TMP*; on other platforms it will look for */tmp*, */var/tmp*, and */usr/tmp*.

> While these are called "temporary files," that is more a reflection of their intended use than of any special treatment they receive. It is up to the program(mer) to delete temporary files and directories that are no longer needed.

Manipulating Paths: os.path

The `os` module contains a submodule, `os.path`, which contains functions for manipulating paths (as opposed to the files and directories at those paths). The following sections outline some of the more important of those functions.

Path components

One of the most frequent kinds of manipulations you will do to file paths is breaking them apart into their components in various ways. The `os` module provides these functions for that purpose:

`os.path.split(`*`path`*`)`
: Returns a pair (***`head, tail`***), where ***`tail`*** is the file part of the path, if any, and ***`head`*** is the directory part of the path, if any (i.e., everything through the last slash is part of ***`head`***, and the rest of ***`path`*** is part of ***`tail`***)

`os.path.dirname(`*`path`*`)`
: Returns the file part of the path (the same as the first element of the pair returned by `split`)

`os.path.basename(`*`path`*`)`
: Returns the directory part of the path (the same as the second element of the pair returned by `split`)

`os.path.splitdrive(`*`path`*`)`
: Returns a pair (***`drive, tail`***), where ***`drive`*** is the drive specification, if any, and ***`tail`*** is the remainder of the path

`os.path.splitext(`*`path1`*`)`
: Returns a pair (***`path2, ext`***); if ***`path1`*** contains one or more periods (`.`), ***`path2`*** will be the part of ***`path1`*** up to the last period and ***`ext`*** the part after the last period; otherwise, the result is (***`path1`***`, ''`)

As we saw, the function `os.listdir(`***`dirpath`***`)` returns a list of the names of all the files and directories that are in the directory at ***`dirpath`***. The result is often hard to read, partly because it's a list, not the formatted output printed by command-line utilities, and partly because it contains a lot of uninteresting files, such as editor backup files, *.pyc* files, and so on.

The first problem can be solved simply by using `print` on each element of the list returned by `os.listdir`. The second problem can be solved by using `os.path.splitext` to filter out files with extensions you're not interested in seeing. These solutions are incorporated into the function defined in Example 6-6, a Filtered Do (see "Filter" on page 122 in Chapter 4).

Example 6-6. Filtered directory listing

```
def filtered_directory_listing(dirpath = '.',
                               ignore_extensions = ('.pyc', '.bak')):
    for filename in os.listdir(dirpath):
        if os.path.splitext(filename)[1] not in ignore_extensions:
            print(filename)
```

Path manipulations

Many programs deal with paths that have no or only a few directory names, but need full paths for certain purposes. The following functions expand or join paths in ways that would be difficult, or at least laborious, to program yourself:

`os.path.abspath(`*`path`*`)`
: Returns a full absolute version of ***`path`***

`os.path.expanduser(path)`
: Returns *path* with an initial ~ or *~username* replaced by the user's home directory

`os.path.join(path1, path2, ...)`
: Returns a path formed by joining the arguments appropriately; roughly similar to the expression `os.sep.join(path1, path2, ...)` (`os.sep` is the platform-specific path separator mentioned a few pages back) except it doesn't add separators after arguments that already end in a separator, and any argument that is an absolute path causes it to ignore the arguments that precede it

The functions in the `os.path` module are important tools. Many programs perform at least a few manipulations on file and directory paths. In addition, you can use the `os.path` functions to define some utilities for your own convenience when using the interpreter. Here are a few examples:

```
def cd(path):
    """Make path the current directory, expanding environment
    variables and a ~ representing the user's home directory"""
    os.chdir(os.path.expandvars(os.path.expanduser(path)))

def merge_ext(path, ext=''):
    """Return path with its extension replaced by ext, which normally
    starts with a period; with no arguments, remove the extension"""
    splitpath, splitname = os.path.split(path)
    return os.path.join(splitpath, os.path.splitext(splitname)[0] + ext)
```

Path information

Often, you just need to test something about a path before doing something with it. Here are three predicates and a function to get the size of a file:

`os.path.exists(path)`
: Returns `True` if there is there a file or directory at *path*

`os.path.isfile(path)`
: Returns `True` if there is there a file at *path*

`os.path.isdir(path)`
: Returns `True` if there is there a directory at *path*

`os.path.getsize(path)`
: Returns the size in bytes of the file at *path*

Example 6-7 shows an example of a Filtered Combine iteration (see "Filter" on page 122 in Chapter 4) in a complete program. The collection is the list of file and directory names in a path; the test is `os.path.isfile`; and what is being combined is the result of a call to `os.path.getsize`. To emphasize how brief well-written code can be, most of the examples in this book omit the imports and the "if main" code at the end. They are included in this example, though, so you can see what it's like to turn a simple function like this into a usable program.

Example 6-7. Calculating the total size of files in a directory

```
"""Print the sum of the sizes of the files in the directory given on the command line
(or the current directory if none is given), in megabytes rounded to 2 places"""

import os.path

def directory_size(path='.'):
    """Sum of the sizes of all files in the directory at path, including
    those beginning with a '.', and ignoring subdirectories"""
    result = 0                                  # identity element
    for item in os.listdir(path):
        if os.path.isfile(item):                # the test
            result += os.path.getsize(os.path.join(path, item))
    return result

if __name__ == '__main__':
    if len(sys.argv) > 2:
        print('Usage: directory_size path')
    else:
        size = directory_size('.' if len(sys.argv) == 1 else sys.argv[1])
        print(round(size/1024/1024, 2), 'Mb', sep='')
```

Another very useful utility would be one that prints a directory structure in hierarchical form. We can define one based on the Recursive Tree Iteration template of Chapter 4 (see "Recursive iterations" on page 128). However, whereas the template assumed the first argument was an entire tree, in this example each call to the function provides just a path. The function gets the tree below that path from a call to `os.listdir`. Example 6-8 shows the code.

Example 6-8. Hierarchical directory listing

```
def dirtree(path='.', ignoredots=True, level=0):
    print_path(path, level)                     # "do something" with tree root and level
    for name in os.listdir(path):               # repeat with the rest of the tree
        subpath = os.path.join(path, name)
        if os.path.isdir(subpath):
            dirtree(subpath, ignoredots, level+1)

def print_path(path, level):
    print(' ' * 3 * level, path, sep='')
```

Filename Expansion: fnmatch and glob

The `fnmatch` and `glob` modules reproduce the wildcard expansion operations of command-line shells. The difference is that `fnmatch` works directly on strings, while `glob` actually goes out to the indicated path to expand wildcards relative to the filesystem content. Both use the command-line matching syntax shown in Table 6-6.

Table 6-6. Wildcard characters for fnmatch and glob

Pattern	Meaning
`*`	Match 0 or more characters
`?`	Match a single character
`[characters]`	Match any of the characters inside the brackets; the characters are not separated (e.g., `'[aeiou]'`)
`[!characters]`	Match any of the characters *except* those inside the brackets

fnmatch

The main functions of the `fnmatch` module are as follows:

`fnmatch.fnmatch(`*`filename`*`, `*`pattern`*`)`
: Returns `True` if the string *`filename`* matches the string *`pattern`*

`fnmatch.filter(`*`filenames`*`, `*`pattern`*`)`
: Returns the list of strings in *`filenames`* that match the string *`pattern`*

> The `fnmatch` module is extremely simple: it treats periods and slashes the same as any other characters. For example, `'.pythonrc'` matches `'*rc'`. This would not be the case on the command line.

Example 6-9 gives an example of using `fnmatch.filter` in a Collection Combine (see Example 4-15, shown in the section "Combine" on page 117) to search a directory hierarchy for all files whose names match a particular pattern. You could use this to locate all files with a particular extension (such as *.fasta*), all files with a particular prefix (such as `gb`), and so on.

Example 6-9. Find files matching a pattern

```
def find_matching_files(startpath, pattern):
    """Return a list of filenames that match pattern in the directory tree starting at startpath"""
    paths = []
    for path, dirnames, filenames in os.walk(startpath):
        for dirname in dirnames:
            if dirname[0] == '.':
                dirnames.remove(dirname)
        paths += [os.path.join(path, filename)
                      for filename in fnmatch.filter(filenames, pattern)]
    return path
```

glob

More useful is the `glob` module. It uses the same pattern syntax but matches against the contents of the filesystem itself, not against a list of strings the way the `fnmatch` functions do. `glob` matches the way the command line does: it ignores files beginning

with a period (unless the pattern begins with a period), and it treats slashes as separating path components.

> The `glob` functions do *not* expand tildes and environment variable names in paths; for that you need `os.path.expandvars`.

The `glob` module can return results either as a list or as a generator, depending on which function you call. The generator version is important for when there are a very large number of filenames matching the pattern, only some of which need to be used. The two relevant functions are:

`glob.glob(`*`pattern`*`)`
: Returns a list of paths (strings) that match ***`pattern`*** (which doesn't necessarily contain wildcards)

`glob.iglob(`*`pattern`*`)`
: Same as `glob.glob`, but returns an iterator instead of a list

For example, to do something to each Python file in the current directory:

```
for filename in glob.iglob('*.py'):
    . . .
```

Shell Utilities: shutil

The `shutil` module provides functions for performing common file operations at a higher level than those provided by `os`—copying and deleting, in particular. There are several kinds of copy operations in the module, but the ones you would normally use are as follows:

`copy(`*`source, destination`*`)`
: Copies the file specified by the path ***`source`*** to the file or directory specified by the path ***`destination`***; if ***`destination`*** is a directory, the file will be copied into it

> Like its command-line equivalent, `copy` will overwrite a file of the same name as the one specified by ***`source`***—either a file by that name in ***`destination`***, if it's a directory, or in the directory containing ***`destination`***, if it's a file.

`copytree(`*`source, destination`*`)`
: Recursively copies the directory ***`source`*** to ***`destination`***; all parent directories in the path ***`destination`*** will be created as necessary and an `OSError` is raised if ***`destination`*** already exists

Moving and removing operations are:

`move(`*`source, destination`*`)`
: Moves the file or directory ***`source`*** to ***`destination`***

`rmtree(`*`path`*`)`
: Deletes the directory ***`path`*** and everything under it

> When programming with these functions you should test your code on directories and files set up for that purpose, so you don't end up unintentionally copying or deleting files.

Comparing Files and Directories

You may be familiar with using a feature of an editor or even a separate application to compare the contents of two files or two directories. Operations like these can have many practical uses in programs, too. For example, you may need to extract the differences between two versions of a program's hourly or daily output, or you might need to know if the results of some BLAST queries you've run are any different from the previous results. Directory comparison is similarly useful: you can compare the contents of directories that contain files produced by programs you are using and extract or report the differences.

File and directory comparison: filecmp

The `filecmp` module provides just two functions, along with a class having only a few methods, but they are powerful and easy to use:

`filecmp.cmp(`*`filepath1, filepath2`*`)`
: Compares the files at ***`filepath1`*** and ***`filename2`*** and returns `True` if their contents are equal

`filecmp.cmpfiles(`*`directorypath1, directorypath2`*, *`filepaths`*`)`
: Compares the files in the list ***`filepaths`*** in the directory at ***`directorypath1`*** with the corresponding files in the directory at ***`directorypath2`***, and returns three lists of file paths:

  `matches`
  : The paths in ***`filepaths`*** to files whose contents were the same in both directories

  `mismatches`
  : The paths in ***`filepaths`*** to files that were in both directories but had different contents

  `errors`
  : The paths to files in ***`filepaths`*** that were not found in both directories or that caused some kind of error when an attempt was made to read them (e.g., because of inadequate user permissions)

`filecmp.dircmp(`*`directorypath1, directorypath2, hidenames`*`)`
: Creates an instance of the class `filecmp.dircmp` to compare the directories at *`directorypath1`* and *`directorypath2`*, with *`hidenames`* a list of names to ignore (defaulting to `[os.curdir, os.pardir]`)

Instances of `filecmp.dircmp` implement the following methods that print fairly elaborate reports to `sys.stdout`:

`report()`
: Prints a comparison between the two directories

`report_partial_closure()`
: Prints a comparison of the two directories as well as of the immediate subdirectories of the two directories

`report_full_closure()`
: Prints a comparison of the two directories, all of their subdirectories, all the subdirectories of those subdirectories, and so on (i.e., recursively)

In addition, many details of the comparisons that the reporting methods print out may be accessed directly as attributes. Their names use the syllable "left" for what was found in *`directorypath1`* and "right" for what was found in *`directorypath2`*. The attributes are:

`left_list`
: The names of files and subdirectories found in *`directorypath1`*, not including elements of *`hidelist`*

`right_list`
: The names of files and subdirectories found in *`directorypath2`*, not including elements of *`hidelist`*

`common`
: The names of files and subdirectories that are in both *`directorypath1`* and *`directorypath2`*

`left_only`
: The names of files and subdirectories that are in *`directorypath1`* only

`right_only`
: The names of files and subdirectories that are in *`directorypath2`* only

`common_dirs`
: The names of subdirectories that are in both *`directorypath1`* and *`directorypath2`*

`common_files`
: The names of files that are in both *`directorypath1`* and *`directorypath2`*

`common_funny`
: Names common to both *`directorypath1`* and *`directorypath2`* but that name a file in one and a directory in the other, along with names of files and directories that caused an error when an attempt to read them was made

`same_files`
: The paths to files whose contents are identical in both *`directorypath1`* and *`directorypath2`*

`diff_files`
: The paths to files that are in both *`directorypath1`* and *`directorypath2`* but whose contents differ

`funny_files`
: The paths to files that are in both *`directorypath1`* and *`directorypath2`* but could not be compared for some reason

`subdirs`
: A dictionary that maps names in `common_dirs` to `dircmp` objects

Showing the differences between two files: difflib

Suppose you are faced with two large FASTA files that you know differ only slightly. You may know how to use your editor, the Unix `diff` command, or another application to get details about the differences between the two files. But what if getting those differences is part of a series of steps your program must perform that at some point involves the difference between the two files. You could use `subprocess.getoutput`, giving as its argument a command you construct, but then you would have to write code to parse the result it returns. What you really should use is Python's `difflib` module. It provides classes for three different kinds of comparisons:

`difflib.SequenceMatcher`
: Finds the differences between two hashable sequences (i.e., strings, tuples, tuples of strings, etc.). It provides the following instance methods:

  `set_seqs(`*`seq1`*`, `*`seq2`*`)`
  `set_seq1(`*`seq1`*`)`
  `set_seq2(`*`seq2`*`)`
  : Set the sequences to be compared.

  `find_longest_match(`*`start1`*`, `*`end1`*`, `*`start2`*`, `*`end2`*`)`
  : Finds the longest sequence common to *`seq1`*`[`*`start1`*`:`*`end1`*`]` and *`seq2`*`[`*`start2`*`:`*`end2`*`]`, returning three values: the position in *`seq1`* where the common sequence began, the position in *`seq2`* where the common sequence began, and the length of the common sequence

  `get_matching_blocks()`
  : Returns a list of triples describing all the matching subsequences of *`seq1`* and *`seq2`* in the form returned by `find_longest_match`

`difflib.Differ`
: Compares two sequences of lines of text, finding both completely different lines and differences within a pair of lines using the following method:

`compare(`*`stringlist1, stringlist2`*`)`

Compares the sequences of strings ***`stringlist1`*** and ***`stringlist2`*** and returns a list of lines showing which are the same in both lists, which appear only in ***`stringlist1`***, which appear only in ***`stringlist2`***, and which appear in both but with slight differences (which are marked in the result returned)

`difflib.HtmlDiff`

Like `difflib.Differ`, but prepares HTML-formatted output according to specified parameters. It provides two methods:

`make_file(`*`stringlist1, stringlist2`*` [, `*`title1`*`=''[,`*`title2`*`='']])`

Returns a string containing the HTML for a table highlighting both the interline and intraline differences between the strings in ***`stringlist1`*** and those in ***`stringlist2`***. (The function's name is a misnomer: it doesn't write to a file, it returns a string.) The optional parameters ***`title1`*** and ***`title2`*** are the strings to use as column headers in the HTML table.

`make_table(`*`stringlist1, stringlist2`*` [, `*`title1`*`=''[,`*`title2`*`='']])`

Works just like `make_file`, except that it only returns the HTML for the difference table itself, not the surrounding text that would make it a complete HTML page.

There are further functions in `difflib` and other methods supported by its three classes. These can provide extensive control over exactly what is matched and how it is reported.

Working with Text

Next, we'll look at some modules for text manipulation. Three address specific needs. A fourth supports reading from and writing to strings instead of files, a feature with wide applicability.

Formatting Blocks of Text: textwrap

The `textwrap` module provides functions for formatting blocks of text. Programming such operations is notoriously tedious and error-prone, so it's good to have these functions available when you need them:

`textwrap.dedent(`*`text`*`)`

Returns a copy of the string ***`text`*** with whitespace common to all of its lines removed; for instance, if every line begins with four or more spaces, the result will be shifted to the left four spaces. While tabs and spaces are both whitespace, this function does not treat them as matching.

`textwrap.wrap(`*`text`*`, [`*`width`*`[, ...]])`

Wraps the string ***`text`*** as a single paragraph, returning a list of lines (without newline characters) of at most ***`width`*** characters (default `78`).

`textwrap.fill(text, [width[, ...]])`
: Wraps the string *text* as a single paragraph, returning a single string containing the wrapped paragraph (including newlines). This is just shorthand for:

```
'\n'.join(textwrap.wrap(text))
```

In addition to *width*, both `wrap` and `fill` accept a large number of keyword arguments for controlling their behavior:

`expand_tabs` *(default* `True`*)*
: If true, use `text.expandtabs` in place of `text`

`replace_whitespace` *(default* `True`*)*
: If true, replace every whitespace character with a space; if `expand_tabs` is false each tab character is replaced with just one space, but if it's true tabs are expanded before whitespace is replaced

`drop_whitespace` *(default* `True`*)*
: If true, eliminate whitespace that appears at the beginning and ends of lines after wrapping, except for whitespace at the beginning of the first line

`initial_indent` *(default* `''`*)*
: Specifies a string to add at the beginning of the first line

`subsequent_indent` *(default* `''`*)*
: Specifies a string to add at the beginning of every line except the first

`fix_sentence_endings` *(default* `False`*)*
: If true, tries to identify sentence endings and ensures that sentences are separated by exactly two spaces

`break_long_words` *(default* `True`*)*
: If true, words longer than *width* will be broken so that no line is longer than *width*

`break_on_hyphens` *(default* `True`*)*
: If true, lines may be broken at hyphens in compound words in addition to being broken at whitespace

The `textwrap` functions `fill` and `wrap` are particularly useful for breaking up a sequence into lines of a fixed length, as when writing a FASTA file. Given a sequence string named `seq`, we can divide it into a list of lines using `wrap` or keep it as a long string but with newline characters inserted at appropriate positions using `fill`:

```
textwrap.wrap(seq, 80)          # break up seq to a list of 80-line strings
textwrap.fill(seq, 80)          # insert newlines every 80 characters
```

Of course, like all string manipulations in Python and its library modules, `textwrap.wrap` returns a new string.

String Utilities: string

An assortment of string-related values, functions, and classes are collected in the `string` module. This module has nothing to do with the implementation of the built-in `str` type; it provides strings that describe kinds of characters and a *template* facility. Useful string module values include:

`string.whitespace`
: Space, tab, line feed, return, form feed, and vertical tab

`string.punctuation`
: Punctuation characters as defined in the local character set

`string.digits`
: `'0123456789'`

`string.hexdigits`
: `'0123456789abcdefABCDEF'`

`string.ascii_lowercase`
: `'abcdefghijklmnopqrstuvwxyz'`

`string.ascii_uppercase`
: `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`

`string.ascii_letters`
: `string.ascii_lowercase` + `string.ascii_uppercase`

`string.printable`
: `whitespace + punctuation + digits + ascii_letters`

The module has one function that will occasionally be useful:

`string.capwords(`*`string`*`)`
: Returns a new *`string`* formed by splitting *`string`* at whitespace, capitalizing each of the words, and joining them. Its definition is:

```
def capwords(s, sep=None):
    return (sep or ' ').join([x.capitalize() for x in s.split(sep)]
```

The module's template facility is easier to use than `str.format` with large bodies of mostly constant text (e.g., a generic web page with placeholders for a few details). Substitution is done according to a dictionary. Wherever a placeholder of the form `$`*`key`* appears in the template, the value of *`key`* in the dictionary is substituted. Keys must be valid identifiers (i.e., names you could give to values, functions, etc.). `$$` is replaced by a single `$`. In contexts where `$`*`key`* is followed immediately by characters that could be part of a key, `${`*`key`*`}` is used instead.

A template is created as an instance of the class `Template`, with the template string given as an argument. To make the substitutions on the template instance's string, the method `substitute` is called on the instance with a dictionary as its argument. If any of the keys are missing from the dictionary, `substitute` raises a `KeyError` exception. The

method `safe_substitute` is similar, but instead of raising an exception for missing keys it leaves the original placeholder intact. There is no provision for deleting placeholders that aren't keys in the dictionary.

Example 6-10 shows a function that demonstrates the use of templates for a simple form-letter facility. It assumes there are a number of files with similar placeholders. The function's parameters are a subject string, the name of the file containing the template, a substitutions dictionary, and a list of email addresses. The function creates a template from the contents of the designated file, makes substitutions using the dictionary, and for each email address calls the function `sendmsg`, defined in Example 6-3.

Example 6-10. Using the string template facility to email a form letter

```
def broadcast(subject, filename, substitutions, receivers):
    with open(filename) as file:
        message = string.Template(file.read()).substitute(substitutions)
    for receiver in receivers:
        sendmsg('your-program',
                receiver,
                subject=subject,
                msg=message
                )
```

Comma- and Tab-Separated Formats: csv

Comma-separated values (CSV) files are a widely used "interchange" format, especially in conjunction with spreadsheet and data-collection applications. The term actually applies to a variety of text field conventions, including tab-separated. The exact rules for representing a field's value are not standardized and can vary in frustrating ways from one application to another. Python's `csv` module hides this variability from you, allowing you to focus on the real purpose of your program.

Very briefly, the module contains the functions `csv.reader` and `csv.writer`. It also includes facilities for defining other formats and for more complicated reading and writing operations, but we won't cover those details here. Here are the basics:

`csv.reader(`*`source`*`[, dialect='excel'])`
: Returns an iterator for the lines of *`source`* (which can be any iterable that produces strings; if a file, it must already be open).

> When opening a file to use with `csv.reader`, it is necessary to provide a keyword argument in the call to `open` for a parameter that wasn't included in the description of the function in Chapter 3: `newline=''`. This allows the reader to correctly interpret newline characters inside quoted field values in a platform-independent way.

Each step of the iterator (either one cycle of a `for` statement or the result returned by a call to `next`) produces a list of the values in the comma-separated fields of one line.

If `dialect` is `'excel-tab'`, reads the file as tab-separated values; otherwise, reads it as comma-separated values.

`csv.writer(`*`destination`*`[, dialect='excel'])`

Returns an instance of a `csvwriter` for a file—or anything else with a `write` method—that is open for writing.

If `dialect` is `'excel-tab'`, writes tab-separated values; otherwise, writes comma-separated values.

The writer provides the following methods (the "row" in their names refers to the widespread use of CSV files as representations of simple spreadsheets, with each list of fields corresponding to a spreadsheet row):

`writerow(`*`fieldlist`*`)`

Writes the fields of ***`fieldlist`*** to the writer's ***`destination`*** according to the writer's dialect, followed by `'\n'`

`writerows(`*`rowlist`*`)`

Does `csvwriter.writerow` for each row in ***`rowlist`***

String-Based Reading and Writing: io

The `io` module provides the core implementation of Python's input and output facilities, including the definition of `open`. It offers classes for a variety of capabilities. Some of them can be used for both reading and writing, while others are for just one or the other.

`StringIO` is used for reading from and writing to strings, and `ByteIO` is for reading from and writing to `bytearrays`. This allows you to work with data in memory just as you would with a file on the disk, with the same set of methods (such as `readline` and `writeline`). You can also provide instances of these classes as the optional `file` parameter in calls to `print`.

To write to a string, you create an instance of `io.StringIO`, generally with no arguments. That object contains an internal buffer, which is the destination for the strings written to it. The method `getvalue` obtains a string based on the contents of that buffer. You can call it at any time, even multiple times, until `close` is called for the instance. (You can either call `close` explicitly in a `finally` clause of a `try` statement, or simply use a `with` statement, which will call `close` for you the way it does with files.)

TEMPLATE

String Output

```
import io
with io.StringIO() as dest:          # create new StringIO
    write to dest using methods such as write and writelines
    result = dest.getvalue()         # get the string from dest
```

Instances of `io.StringIO` are also used for reading from strings. The difference is that when you create the instance you provide a string as an argument. Subsequently, the `io.StringIO` instance can perform the usual input operations, such as `readline` and `readlines`.

TEMPLATE

String Input

```
import io
with io.StringIO(string) as src:
        # string is the StringIO source
    read from src using methods such as readline and readlines
```

With `io.StringIO`, you can use a string as the source in place of a file in code such as the definition of `get_items` shown in various earlier examples. Where a function uses `open` and passes the open file to `get_items`, it could just as well create an `io.StringIO` from a string and pass that instance to `get_items`. This would work with most of the examples in Chapters 3 and 4 that used `readline` to parse the contents of a file.

Persistent Storage

Data stored in text files is adequate for many applications. However, as the quantity and complexity of the data increases, it becomes too cumbersome to rely on formatted text. This is especially true when an application selectively accesses small amounts of data from a much larger store. For these cases, Python provides three *persistent storage* modules that use special binary files to store objects more like the way they are represented in memory. These mechanisms make access to external data much faster and much easier to program.

Persistent Text: dbm

The simplest form of persistent storage is essentially a file-based dictionary whose keys and values are both text. This is implemented by the `dbm` module, which provides the following function for opening database files:

`dbm.open(filename[, flag])`
: Returns an object providing access to the database in *filename*; *flag* must be one of the following:

  `'r'`
  : Opens an existing database for reading (the default)

  `'w'`
  : Opens an existing database for reading and writing

  `'c'`
  : Opens a database for reading and writing, creating it if it doesn't exist

  `'n'`
  : Opens a new, empty database for reading and writing, replacing it if it already exists

Only one program at a time can have a `dbm` database open (with an exception shown for the `gnu` variation discussed momentarily). Table 6-7 shows the operations and methods supported for an object returned from `dbm.open`.

Table 6-7. Core dbm operations and methods

Operation/method	Description
`db[str1] = str2`	Stores *str2* as the value of key *str1* (as `bytes`)
`db[str1]`	Returns the value of key *str1* (as `bytes`)
`del db[str1]`	Deletes the entry for the key *str1*
str1 in db	Returns true if db contains the key *str1*
str1 not in db	Returns true if db does not contain the key *str1*
`db.keys()`	Returns a list of all the keys in db

There are three submodules of `dbm`, each of which implements the core functionality in a different way. When you create a new file with `dbm.open` it determines which submodule's `open` function to use based on what facilities are available in the operating system environment.† When `dbm.open` is called on an existing file, it determines which of the implementations created it and calls the corresponding submodule's `open` function. (You can tell by the type of the object returned which module was used; you can also specify which to use by importing a submodule and calling its `open` function.) The submodules are:

`dbm.ndbm`
: An implementation based on the Unix (`n`)`dbm` facility

† Python running on Windows will have only `dbm.dumb`. Otherwise, `dbm.ndbm` should be present as well. When installed through a Linux package manager, Python may have `dbm.gnu`. The `dbm.gnu` implmentation will also be available in Python built from its source files with the `gdbm` library installed.

`dbm.gnu`
: An implementation based on GNU `gdbm`

`dbm.dumb`
: A pure Python implementation

Beyond their implementation of the core operations shown in Table 6-7, the submodules differ somewhat. The current implementation of `dbm.dumb.open` ignores its *`flag`* parameter: files are always opened for reading and writing and created if they don't already exist. The `ndbm` implementation provides two additional methods that are just like the ones with the same names provided by `dict`: `get` and `setdefault`.

The `gnu` variation is the richest of the three. The *`flag`* parameter of `gnu.open` can have additional characters that specify further details of how the database file will be used:

`'f'`
: Opens an existing database in "fast mode"—individual changes to the database are not immediately written to the file but rather are buffered for efficiency, similar to the way the buffered output stream `sys.stdout` works

`'s'`
: Opens an existing database in "synchronized mode"—each individual change is immediately written to the database file, similar to the way the unbuffered output stream `sys.stderr` works

`'u'`
: Doesn't lock the database, allowing other programs and other users to access its contents while your program has it open

The `gnu` submodule also provides some additional features. The method `reorganize` is used to compact the database file after many items have been deleted. If the database has been opened with `'f'` included in the *`flag`* argument, the method `sync` forces accumulated changes to be written out to the file.

The `firstkey` and `nextkey` methods allow you to iterate using the keys of a `gnu` database. Start with a call to `firstkey`, which returns a key. Then call `nextkey` with the first key to get the next one. Repeat the process with the result of the previous call to `nextkey` until `nextkey` returns `None`.

The Rebase data reader example at the end of Chapter 4 constructed a dictionary whose keys were the names of restriction enzymes and whose values were recognition sequences. It included functions to write the parsed data to a file in a simple format and to read the data from that file into a dictionary. With a few small changes, we could convert that program to use `dbm` instead of a dictionary to store its data. This would mean the data would be loaded only once; subsequently, individual entries could be accessed directly from the database file, and programs that use the database wouldn't need the reading and writing functions at all.

The datafile used for this example is fairly small, so in practice switching to `dbm` wouldn't make much of a difference. However, if the datafile were much larger it could be significantly more efficient to load the data just once and then access it directly from the database. Even if the file is relatively small, it can be advantageous for other programs to use `dbm` to access the data so that they won't need the original parsing code. (As always, we want a given piece of code confined to one place, especially one as complex as the parsing code from Chapter 4's examples. In fact, it would be a good idea in this case to have the code that loads the database in a separate file from the code that uses it.)

Example 6-11 shows the functions from that example for loading the data into the table, with modifications to use `dbm`. The functions that parse the original file will be unaffected by the switch to `dbm`, but they will be used only once, to load the database. Note that `dbm` files must be closed, and they must be closed explicitly—they don't support the automatic closing feature of `with` statements.

Example 6-11. Loading Rebase data into a dbm database

```
import dbm

def load_enzyme_data(data_filename, dbm_filename):
    try:
        table = dbm.open(dbm_filename, 'n')           # 'n' for new database
        with open(data_filename) as datafile:
            load_enzyme_data_into_table(datafile, table)
    finally:
        table.close()

def load_enzyme_data_into_table(datafile, table):
    line = get_first_line(datafile)
    while not end_of_data(line):
        key, value = parse(line)
        store_entry(table, key, value)
        line = get_next_line(datafile)
    # return table    not needed

def store_entry(table, key, value):
    table[key] = val                          # table is an open dbm file, but [ ] unchanged
```

After loading the data into the database by running this program, individual entries would be accessed as follows:

```
>>> table = dbm.open(dbm_filename, 'r')
>>> table['AatII']
b'GACGT^C'
>>> table.close()                         # can do repeated reads before closing
```

Notice that the data returned is a `bytes` object, not a string. The database stores both keys and values as `bytes`. Strings are automatically converted into `bytes` when stored in the database, but not when they are retrieved.

To convert a `bytes` object to a `str` you need to call its `decode` method. If you want to use an encoding other than the default, you should convert your keys and values to `bytes` explicitly when storing them into the database. To convert a string `s` to `bytes` for a chosen value of *`encoding`*, (e.g., `'utf-8'`), call `s.encode(`*`encoding`*`)` when storing it into the database. To convert a `bytes` *`b`* to a string using a specific encoding, call `b.decode(`*`encoding`*`)`.

Persistent Objects: pickle

Using disk files to store bytes is straightforward, since bytes are exactly what files contain. All the `dbm` module adds to the raw bytes is a bit of structuring so it can efficiently access values by their keys, the way dictionaries do. (Some versions of `dbm` use a separate file for an index into the file.) This simple mechanism is fine if all you need is a persistent dictionary with keys and values stored as bytes. Storing anything more complicated presents a number of problems, though, some of them quite deep.

The Python module `pickle`[‡] solves some of those problems. It encodes objects into byte representations from which the objects can be re-created. This facility can handle self-referential data structures, references to types and functions, large objects, and objects with intricate internal structures. The module's functions read and write byte-encoded representations of objects, but you don't need to know anything about those representations—the module takes care of all the details for you.

Pickling is strictly byte-based; therefore, files used to write and read pickled objects must be opened in binary mode. This doesn't affect the programs: unlike `dbm`, `pickle` restores objects to their original types. In particular, stored strings are restored with type `str`, not `bytes`.

When an object is encoded, any object that is the value of one of its attributes will also be encoded, along with references to classes and functions. Therefore, pickling one object may actually result in the encoding of many objects. It is often convenient to pickle an entire list or dictionary, which in turn pickles all the list elements or dictionary keys and values.

The `pickle` module's core functions are:

`pickle.dumps(`*`object`*`)`
: Produces a byte-encoded representation of *`object`* that `pickle.loads` can use to create an equivalent object

‡ Other languages, libraries, and books use other terms for what Python calls "pickling," including "marshaling," "serializing," "flattening," or "persisting."

`pickle.loads(`*`bytesinstance`*`)`
: Re-creates the object represented by ***`bytesinstance`***, such as that returned by `pickle.dumps`

`pickle.dump(`*`object, bytestream`*`)`
: Pickles an object and writes it to ***`bytestream`***, which must be a binary write stream such as a file opened with mode `'wb'` or an instance of `ByteIO`; more than one object may be pickled and written to the same byte stream

`pickle.load(`*`bytestream`*`)`
: Reads an object from ***`bytestream`***, which must be a binary read stream such as a file opened with mode `'rb'` or an instance of `ByteIO`, and unpickles it; ***`bytestream`*** may contain a series of pickled objects, in which case each call to `pickle.load` reads just one

TEMPLATE

Using pickle

To save an object using `pickle`:

```
with open(filename, 'wb') as file:
    pickle.dump(object, file)
```

To load an object from a `pickle` file:

```
with open(filename, 'rb') as file:
     result = pickle.load(file)
```

An important use of pickling, though we won't demonstrate it, is for communicating objects across networks. There is an example in Chapter 9 (see "Server fundamentals" on page 338) that demonstrates sending simple text between a client and a server. Much more complex architectures can involve cooperating processes on multiple machines rather than a simple client/server configuration, in what's known as *distributed computing*. Typically, these processes must exchange more complex data. If the processes are running Python programs they can send (copies of) groups of objects among each other by pickling them and sending the resulting text, which can then be unpickled by one or more receivers.

Keyed Persistent Object Storage: shelve

One of the limitations of pickling objects to files is that the code loading the objects must know what to do with what it loads. This isn't a problem if the file contains just one object, including the case where what was pickled was a single collection. With many separate objects pickled to a file, it would certainly be possible to retrieve them one at a time in a loop, collecting them into a list or dictionary. However, this would not be a useful approach in situations where only one of many objects is needed.

Pickling a collection of objects also has its own problems. A dictionary with many thousands of list values may be used by a number of small applications. Unpickling the entire collection of objects each time a program starts could lead to long delays, in addition to wasting a great deal of memory space when only a few instances of the classes are actually needed. The same problems reappear if new instances are created or old ones changed: it will be necessary to repickle the whole collection.

The solution to these problems is to pickle each object individually, rather than pickling whole collections of objects, and to identify each one by a name. Instead of pickling and unpickling the entire dictionary, this approach will allow an application to access just the values it needs using their names. As with so many other things in Python, the way to do this is with a dictionary mechanism. The module `shelve` combines the keyed-storage aspect of `dbm` with the object persistence capabilities of `pickle`. Keys must be instances of `str` (not `bytes`). The values may be anything that can be pickled. Like `dbm`, the `shelve` module provides a function for opening database files:

`shelve.open(`*`filename`*`[, `*`flag`*`='c'])`
: Opens *`filename`* as a persistent dictionary (i.e., a dictionary whose keys are strings and whose values are pickled objects); *`flag`* has the same options as for `dbm.open` (see "Persistent Text: dbm" on page 243)

An opened "shelf" acts like a dictionary, with one critical distinction: modifying a value obtained from a shelf does not affect the value stored in the shelf file. The only actions that affect the file are assignments and deletions—i.e., *`db`*`[`*`key`*`]=`*`value`* and `del db[`*`key`*`]`. If you want to modify a value that is in the file, you must explicitly use an assignment statement with the modified value.

TEMPLATE

Using shelve

Assignment of a shelf entry does the pickling for you, so this module is very easy to use.

```
db = shelve.open(filename, 'c')
db['key'] = value          # store value in the shelf, naming it key
db['key']                  # get the value with name key
del db['key']              # remove the entry with name key
db.close()
```

Debugging Tools

Debuggers are facilities that allow you to investigate the state of your program when its execution has stopped. You can use them to determine the sequence of function and method calls that led to the current place in the program, look at the values of local names, execute assignment statements to rebind local names, and "step" through your code to observe what happens.

There are two kinds of Python debugging facilities: the `pdb` module and tools built into IDEs, including IDLE.

The Python debugger: pdb

Whether from the command-line interpreter, IDLE's Python Shell window, or even a Python file, importing the module `pdb` gives you access to some highly useful debugging tools. The debugger uses its own read-evaluate-print loop, with the prompt `(pdb)` and some different behavior than is found at Python's usual top level:

- In general, you can type anything to the debugging prompt that you can type to Python, with the same effect.
- In addition, there is set of *commands*—not function calls—that you can type to perform various actions specific to debugging.

The `pdb` module provides three functions for entering the debugger:

- Immediately after a traceback, before doing anything else, call `pdb.pm0` ("postmortem"). That will put you into the `pdb` command loop, positioned at the location of the error that caused the traceback.
- Put a call to `pdb.set_trace0` in your code. When Python executes that line, it enters the `pdb` command loop.
- Call `pdb.run()`, giving it a string as an argument. The string would be something you would type to the interpreter. The debugger pauses immediately, and then you can step through your code.

You can also run the debugger on a Python file without importing the module by typing the following at the command line:

```
python3 -m pdb file.py
```

This will enter the command loop before the program even starts running.

> The debugger is a tool that warrants continual experimentation and learning. As you become more comfortable with what you learn initially, you can explore more powerful techniques and advanced commands.

Following are descriptions of the basic commands of the `pdb` command loop; you can type just their first letter or the entire word. First we'll look at commands for getting information and quitting:

`h`*(*`elp`*) [command]*
: Without an argument, prints the list of available commands. With an argument, prints help about that command. If the argument is `pdb`, prints the full documentation.

`w`*(*`here`*)*
: Prints a full "stack trace," with the most recent frame at the bottom and an arrow showing the current context for other commands. This is like what you see in a traceback, but it shows the entire stack, not just the last few entries.

`q`*(*`uit`*)*
: Quits the debugger.

The next set of commands are for controlling execution:

`n`*(*`ext`*)*
: Executes code until the next line of the current function is reached.

`s`*(*`tep`*)*
: Executes the current line, stopping at the next executable line. If the current line is a call to another function, this "steps into" that other function, whereas `next` would not.

`r`*(*`eturn`*)*
: Continues execution until the current function returns.

`c`*(*`ontinue`*)*
: Continues execution.

`run` *[`args...`]*
: Restarts the program. If `args` are provided, uses those as the value of `sys.argv`.

The commands you'll use to view values are:

`a`*(*`rgs`*)*
: Prints the value of the current function's arguments.

`p`*(*`rint`*)* *`expression`*
: Prints the value of *`expression`*. Usually this is not needed, since you can just type the expression to get its value; using `p` is necessary only when the expression is something that looks like a `pdb` command (including ones not shown here).

`pp` *`expression`*
: Prints *`expression`* using `pprint.pprint`.

`!`*`statement`*
: Executes *`statement`*. Usually the exclamation point is not necessary, since you can just type a statement to execute it; using `!` is necessary only when the beginning of the statement looks like a `pdb` command.

The expressions evaluated by `p`, `pp`, and typing an expression, as well as the statements executed with `!` or simply by typing them, are evaluated as if they had been in the code of the current stack frame. That means that arguments to the corresponding function and local names it has bound will be available. Typically a stack frame corresponds to a function definition, but comprehensions and exception handlers also create stack frames that will be visible in the debugger. You can change the current focus, or context, of the stack by moving up or down within it, one frame at a time, using these commands:

`d`*(own)*
: Moves down to the next more recent frame

`u`*(p)*
: Moves up to the next older frame

Moving around in the stack does not change its *contents*—that is, the state of the program is not changed; what it changes is the *context* in which debugging commands are executed and expressions evaluated. In particular, moving up to an older frame doesn't remove the more recent ones from the stack. Moving around the stack lets you investigate the values names have in different contexts without disturbing the state of the computation.

Another key feature of debuggers is the ability to set breakpoints. A breakpoint can be set at a particular line of code or in a specific function. When execution reaches that point in the code, the `pdb` command loop is entered. This is something you should eventually explore. More about this, and the rest of the `pdb` features not covered here, can be found in the `pdb` library module documentation (*http://docs.python.org/py3k/library/pdb.html* or the equivalent on your computer).

> The `pdb` loop operates in a different namespace from the top-level Python loop. Some names that are defined at the top level are not defined inside `pdb`, and names assigned inside `pdb` are assigned in the namespace of its current stack frame, not the top level, so the assignment will not survive exiting `pdb`.

The IDLE debugger

IDLE has its own debugger. You open it with the Debug command that replaces the Run Module command when the Python Shell is the current window. When you do so, you'll see a `[DEBUG ON]` message in the shell window. Naturally, you'll see a `[DEBUG OFF]` message when you select the Debug menu command again.

> You can still use `pdb` in IDLE, either by calling `pdb.set_trace` from your code or by calling `pdb.pm` in the interpreter immediately after a traceback.

The IDLE debugger works on similar principles to `pdb`, but it has a simple graphical user interface. There is a Stack Viewer window and a Debug Control window. It's difficult to describe exactly how best to use these, but experimentation will be rewarded. Even if you don't know quite what to do to take advantage of the contents of these windows, open them and get used to seeing what's in them. That will help you use them later, when you need them.

Tips, Traps, and Tracebacks

Tips

- Before writing complicated functions to do things with files and their contents, investigate whether command-line facilities are already available to do what you want. If they are, you can just use `subprocess.call` to invoke them.
- *It is extremely important that you understand where Python looks for modules you want to import (including your own)*. Keep these points in mind:
  - —Import statements look for modules in each directory in the list `sys.path`. The system installation gives `sys.path` an initial value. Packages you obtain from outside the Python installation may also add to that path as part of their installation, as may your own programs.[§]
  - —When you open IDLE by double-clicking a file, the file's directory is added to the front of `sys.path`.
  - —Different IDEs on different platforms have different ideas about the "default directory" for creating, opening, and saving files. Pay attention and learn what yours is doing so that you don't save files in strange places—into the Python installation directories, in particular.
- If you get an error that a file your program is trying to open can't be found, first check that you have turned off the system feature that hides some file extensions, as described on page XXX. The next thing to do is use `os.getcwd()` to confirm that Python's current directory is what you think it is. Finally, use `os.listdir('.')` to see what files Python thinks are in the current directory. The Python installation includes extensive, well-written, helpfully organized, and nicely formatted documentation:
  - —You can find the documentation online at *http://docs.python.org/py3k*.
  - —You can open the documentation from IDLE by selecting the Python Docs command from the Help menu.
  - —On Windows, the documentation is installed in the Python installation directory as a *.chm* file in the *doc* subdirectory.
  - —On OS X, the documentation is a set of HTML files deep inside */Library/Frameworks/Python.framework*; after opening it from IDLE, you can bookmark the URL in your browser.
  - —Bookmark the link to the Python library documentation in your browser. You will use that starting point far more than any other. If you can find the

§ To modify `sys.path`, you must of course first import `sys`; even though its contents are built into Python, until you import it the names it assigns and defines are not in the namespace of the interpreter or a program you are executing from the command line. The same goes for other frequently used system modules, such as `os`.

documentation directory on your computer, link to the *library/index.html* file it contains, so you can work offline. If you can't find it, you can link to its online location at *http://docs.python.org/py3k/library/index.html*.

- Documentation can be downloaded in HTML, PDF, or text format from *http://docs.python.org/py3k/download.html*. After downloading, you can install the files anywhere you'd like.
- At the top- and bottom-right corners of every page of the HTML documentation are links to a module index and to a comprehensive index. The pages those link to are invaluable both for finding what you are looking for at the moment and noticing new things to explore.
- If you are fond of browser windows with multiple tabs, it's a good idea to open tabs to the following pages in the documentation directory—or the online equivalent—and then save the set as a bookmark folder that you can open all at once (note that the first two files are in the same directory as the *library* directory, not underneath it):

  modindex.html
  : The global module index

  genindex.html
  : The general index

  library/index.html
  : Despite the file's name, a table of contents for the entire library documentation

  library/functions.html
  : Documentation of all built-in functions

  library/stdtypes.html
  : Documentation of all built-in types, including their operations and methods

Traps

- It's easy to forget the exact name of the `os.getcwd` function, because it's not like its command-line equivalent, `pwd`.
- If you get an `ImportError` in the interpreter, make sure the current directory is what you think it is. Use `os.getcwd()` to check and `os.chdir(`*path*`)` to change it if necessary.
- Most of the time, the function you want will be in `os.path`, not `os`. If you get a `'module' object has no attribute` error, don't assume you used the wrong function name—check that you did write `os.path`, not just `os`.

Tracebacks

Some of the errors you might encounter when using the utility modules described in this chapter include:

`OSError`
: The functions in `os` raise `OSError` when given arguments of the wrong type, paths that aren't valid (e.g., a file that doesn't exist), or paths to files or directories that your program doesn't have the privileges necessary to read.

`smtplib.SMTPConnectError`
: An attempt to connect to the SMTP server failed.

`smtplib.SMTPAuthenticationError`
: Authentication with the password and username provided failed.

`socket.error`
: A low-level communications failure occurred.

CHAPTER 7
Pattern Matching

This chapter treats an enormously important topic and the Python module that supports it. We'll use its features frequently in examples later in the book. As you gain familiarity with them, you'll be using them increasingly more often. Some patience is required, though: the topic is quite technical, and you can't absorb it all at once. You'll need to come back to this periodically and explore it further. Be ambitious in experimenting with it. This is the only chapter in the book that's like this.

The mysterious topic is string pattern matching using *regular expressions*. The term "pattern matching" should be enough of a hint for you to realize how important this topic is for working with bioinformatics data. It's also important for processing web pages and for many other kinds of work a bioinformatics programmer needs to do. Among other things, restriction enzyme binding sites can be expressed as regular expressions. A carefully constructed regular expression can take the place of many lines of string manipulations, loops, and iterations. Once you've learned to use regular expressions, you'll have a very powerful tool in your hand.

> If you have never encountered regular expressions before—or even if you've used only their most basic features—you should be aware that the topic is rather large, and learning it is not easy. More than most aspects of programming, learning to use regular expressions requires experimentation. Start by using the basics, and whenever you can't do what you want with what you already know, go back to this chapter (or some other documentation) and learn a little more.

Despite how much this chapter gives you to absorb, there are many advanced aspects of regular expressions that aren't covered here at all. One could write an entire book about regular expressions, and in fact several people have.[*] Regular expression facilities are found in the libraries of many languages, as well as in editors, IDEs, and other tools.

[*] See, in particular, Jeffrey Friedl's *Mastering Regular Expressions*, Third Edition, and Jan Goyvaerts and Steven Levithan's *Regular Expressions Cookbook*, both published by O'Reilly.

Unfortunately, the details of regular expressions differ somewhat among different programming languages and editors, and not all resources cover Python's version exactly.

> Working out the details of a regular expression you need for a particular task can be frustrating. You should design and test regular expressions outside of Python before putting them into a program. Some Python IDEs incorporate a regular expression editor and tester, but IDLE does not. An excellent interactive web page for testing regular expressions can be found at *http://re-try.appspot.com* (with a hyphen between "re" and "try"); you should use it often.

Regular expressions are strings used as patterns to perform matching and searching operations on other strings, which we'll call the "targets." The result of a successful operation is a *match object*, which has quite a few fields and methods providing details about the match, such as the position within the target where the pattern was found. If the match or search fails, the result is `None`, rather than a match object.

> Python makes a distinction between *matching* and *searching*, which other languages do not. Matching looks only at the *start* of the target string, whereas searching looks for the pattern *anywhere* in the target. In discussing regular expressions it is very difficult to limit the use of the term "match" to Python's matching operations, and we will use it for both cases.

Fundamental Syntax

Within a regular expression string, certain characters have special significance. The most frequently used are similar to the wildcard characters of command-line file utilities such as `ls` or `dir`. The wildcard characters don't have quite the same meaning in regular expressions as they do on the command line, but the idea is similar.

Backslashes have special significance within regular expressions. Backslashes also have special significance within Python strings in general. Unfortunately, what a backslash means to Python isn't always the same as what it means in a regular expression, and to include a backslash inside a regular Python string requires doubling it as \\ (just as strings can include quotes by preceding them with backslashes).

To help reduce confusion and simplify the use of backslashes in regular expression strings, Python provides a *raw string* syntax: preceding a string with the character `r` or `R` tells Python to ignore the usual special interpretation of backslashes within the string. Whatever backslashes appear in a raw string are just backslashes. *Always use raw strings for regular expressions*. Here is a demonstration of the difference:

```
>>> '\n'
'\n'
>>> len('\n')
```

```
1
>>> r'\n'
'\\n'                            # a slash followed by a newline character
>>> len(r'\n')
2
```

Fixed-Length Matching

Except for characters that have special interpretations, each character in a regular expression matches only that character in the target. We'll start by looking at how restriction enzyme recognition sites can be represented as simple regular expressions.

Literal matches

The recognition site of the restriction enzyme EcoRI is `GAATTC`. The regular expression `'GAATTC'` will match exactly those characters. Since `'GAATTC'` has no special regular expression characters in it, searching for it isn't really any different from using `target.find(pattern)`. Table 7-1 gives a few examples.

Table 7-1. Literal regular expression matches

Pattern	Target	Match position	Search position
GAATTC	GAATTC	0	0
	TTGAATTC	None	2
	AATGTGAATTCT	None	5

Many restriction enzymes recognize sites with more flexibility than a literal match. For instance, EcoRII recognizes either `CCAGG` or `CCTGG`. We don't really need regular expressions to look for any of several substrings: we could easily write functions for matching or searching that do this, as in Example 7-1.

Example 7-1. Functions for multiple match and search

```
def multi_match(target, patterns):
    """Return True if target begins with any of the strings in patterns"""
    for pattern in patterns:
        if target.startswith(pattern):
            return True

def multi_search(target, patterns):
    """Return the first position in target where any of the strings in patterns is found"""
    return min([target .find(pattern)
                for pattern in patterns
                if target.find(pattern) >= 0])
```

Then we could write this:

```
multi_match(sequence, ('CCAGG', 'CCTGG'))
```

to see if `sequence` begins with either of EcoRII's recognition sites, and this:

```
multi_search(sequence, ('CCAGG', 'CCTGG'))
```

to find the first position matched by either site.

The enzyme DsaI recognizes any of `CCGCGG`, `CCGTGG`, `CCACGG`, or `CCATGG`. To use these functions for this enzyme's recognition sites we would have to provide a list of four patterns. SecI recognizes any site of the form `CCNNGG`; that means we'd have to include 16 sequences in the list of patterns for those sites. This would get out of hand pretty quickly. For instance, CjuI recognizes `CA`, followed by `C` or `T`, followed by any five bases, followed by a `G` or an `A`, followed by `TG`. This is 2*4*4*4*4*4*2 = 4,096 possibilities!

Character sets

The simplest tool provided by regular expressions to manage this "combinatorial explosion" of possibilities is the ability to specify a *character set* rather than a single character. A character set consists of one or more characters enclosed in square brackets. Each character set in the pattern corresponds to exactly one character in the target: if none of the characters in the set matches that character in the target, the pattern is not a match. A regular expression for DsaI's site would be:

```
'CC[GA][TC]GG'
```

SecI's sites can be represented as:

```
'CC[TCAG][TCAG]GG'
```

And CjuI's can be represented as:

```
'CA[CT][TCAG][TCAG][TCAG][TCAG][TCAG][GA]GG'
```

Character sets can contain character *ranges* in addition to individual characters. A range consists of two characters separated by a minus sign (`-`). This is equivalent to including the first character, the last character, and every character in between in the square brackets. For example, `'[A-Za-z_]'` would match an underscore or any uppercase or lowercase ASCII letter.

Characters that have special meanings in other regular expression contexts *do not* have special meanings within square brackets. The only character with a special meaning inside square brackets is a `^`, and then only if it is the first character after the left (opening) bracket. In that case it negates the character set, meaning the set matches any character *except* the ones specified inside the square brackets.

If you want to include a right bracket in a character set, you have to make it the first character; otherwise, it will be interpreted as the end of the character set. Similarly, if you want to include a minus sign in the character set you have to make it the first character, or it will be interpreted as showing a range.

You can also preface a `^`, `]`, or `-` with a backslash. While backslashes are used for a lot of things in regular expressions, they are always used with at least one other character, and technically are not one of the characters with "special meanings." To include a backslash in a regular expression—in a character set or by itself—you must use two

backslashes. The usual backslashed characters, such as `\t` for tab, can also be included in character sets.

Within a regular expression, a period matches *any* character. In effect, it represents a universal character set. Table 7-2 gives some examples.

Table 7-2. Character sets in regular expressions

Pattern	Matches
`[ACTG]`	One DNA base character
`[A-Za-z_]`	One underscore or letter
`[^0-9]`	Any character *except* a digit
`[-+/*^]`	Any of `+`, `-`, `/`, `*`, `^`; `^` does not negate the others because it is not the first character in the set
`[0-9\t]`	A tab or a digit
`.`	Any character

Character classes

For convenience, the regular expression syntax offers some notation for *character classes*. These are similar to character sets in that they represent any of a certain group of characters. In fact, a character class can even be included in a character set, the way a range can. Character classes are simply convenient abbreviations for commonly used character sets and ranges that might be included as part of character sets. Table 7-3 lists Python's regular expression character classes.

Table 7-3. Character classes in regular expressions

Character	Matches
`\d`	Any digit
`\D`	Any nondigit
`\s`	Any whitespace character
`\S`	Any nonwhitespace character
`\w`	Any character considered part of a word
`\W`	Any character not considered part of a word

Boundaries

We've already seen in previous chapters that Python makes it very easy to treat a text file as a collection of lines. Several of the chapters that follow this one include examples of regular expressions used in processing plain text. These will include sequence data, protein data, web pages, and text in structured formats.

There is regular expression notation to indicate the beginning or end of a target, the beginning or end of a line within a target, and word boundaries. These *boundary*

indicators are a bit unusual in that they don't actually match any characters: they just indicate positions within the target. They're listed in Table 7-4.

Table 7-4. Boundaries in regular expressions

Character	Matches
^	The start of a line or the beginning of the pattern
$	The end of a line or the end of the pattern
\A	The start of the pattern only
\Z	The end of the pattern only
\b	The boundary between a word and nonword character or vice versa
\B	Anywhere except the boundary between a word and nonword character or vice versa

Table 7-5 gives a few examples of how boundary indicators are used in regular expressions.

Table 7-5. Line beginnings and endings in regular expressions

Pattern	Matches
^CG	A target line or string starting with CG
TATATA$	A target line or string ending with TATATA

Variable-Length Matching

The real power of regular expressions comes from the ability to specify that certain parts of a pattern can be repeated to match the target.

Repetition

Repetition is specified using the notation shown in Table 7-6. Each of these repetition notations refers to the regular expression that directly precedes it. A single letter by itself is a regular expression, as are a character set and a period.

Table 7-6. Repetition characters in regular expressions

Character	Matches
?	Zero or one repetitions of the preceding regular expression
*	Zero or more repetitions of the preceding regular expression
+	One or more repetitions of the preceding regular expression
{*n*}	Exactly *n* repetitions of the preceding regular expression
{*m*,*n*}	Between *m* and *n* (inclusive) repetitions of the preceding regular expression

Using the brace notation, we can collapse the repeated elements of recognition sites into very brief expressions. SecI's sites can be represented as `'CC[TCAG]{2}GG'`, CjuI's 4,096 recognition sequences can be represented as `'CA[CT][TCAG]{5}[GA]GG'`, and the sequences XcmI recognizes can be expressed as `'CCA[TCAG]{9}TGG'`—that's 262,144 possibilities represented as a 15-character regular expression!

Restriction enzyme recognition sites are fixed-length sequences. Variation in what sequences a particular enzyme recognizes arises from positions that can match two, three, or four bases instead of just one. While the number of possible combinations within a fixed-length sequence might be enormous, the number of bases to be matched is fixed for any restriction enzyme.

However, regular expressions are far more general than that. Except for `{m}`, the repetition characters produce regular expressions that can match targets of different lengths. We'll need more general kinds of targets to match against, so for now we'll use arbitrary base and amino acid sequences. Table 7-7 gives some examples.

Table 7-7. Repetition characters in regular expressions

Pattern	Matches
`CC[TCAG]{2}GG`	CC, followed by any two DNA bases, followed by GG
`(TA){3,8}`	Between three and eight repetitions of TA, inclusive
`[GC]*`	Zero or more Gs and Cs (in any combination)
`A+`	One or more As
`AT?AA`	AAA or ATAA only

Greedy Versus Nongreedy Matching

The repetition-matching characters listed in Table 7-7 match the maximum possible portion of the target, provided the rest of the regular expression matches the rest of the target. This is often referred to as *greedy matching*. But much of the time—perhaps most of the time in bioinformatics—greedy matches absorb a far larger portion of the target than was intended.

Suppose you want to search an RNA string for the first occurrence of the polyadenylation signal `AAUAAA`. The regular expression `r'[UCAG]+AAUAAA[UCAG]+'` will match from the beginning of the string to the *last* occurrence of `AAUAAA`. That is because the initial `[UCAG]+` grabs as much of the string as it can, which, since it includes all possible bases, is nearly the entire sequence. All that remains following that part of the match is the last occurrence of `AAUAAA` and one or more bases after it.

Adding a question mark after one of `*`, `+`, or `?` changes the behavior to minimal possible matching, or *nongreedy matching*. The nongreedy matching characters are listed in Table 7-8. To return to the example of trying to find the first `AAUAAA` in an RNA sequence, adding a question mark after each of the plus signs in the pattern will change it so that

it matches the *first* occurrence instead of the last. The pattern would be `r'[UCAG]+?AAUAAA[UCAG]+?'`.

Table 7-8. Nongreedy regular expression repetition characters

Characters	Matches
`*?`	Zero or more repetitions of the preceding regular expression, nongreedily
`+?`	One or more repetitions of the preceding regular expression, nongreedily
`??`	Zero or one repetitions of the preceding regular expression, nongreedily (rare)

The concept of nongreedy matching is a confusing one. When a pattern that contains repetition characters is matched against a target string, there are often many different ways the pattern can match. For example, the pattern `r'[TA]+[UTAG]*` could match the target `'TATATATA'` by matching `[TA]+` against all four `TA`s and `[UTAG]*` against no characters, `[TA]*` against three `TA`s and `[UTAG]*` against the final `TA`, and so on.

The definition of greedy matching is that each variable portion of the pattern will match against as *much* of the target as it can, as long as the rest of the pattern can match the rest of the string. In nongreedy matching, each variable portion of the pattern will match as *little* of the target as possible as long as the rest of the pattern can match the rest of the string.

Grouping and Disjunction

A portion of a regular expression can be enclosed inside parentheses. (To use `(` or `)` as a literal, either precede it with a backslash or put it inside square brackets.) This has much the same effect as using parentheses in algebraic expressions. A repetition character that follows a parenthesized regular expression indicates the repetition of the expression inside the parentheses.

Parentheses do more than just group parts of a regular expression, though. The parts of the target that match groups in a regular expression are assigned numbers in sequence (starting with 1). One part of a regular expression can refer to an earlier part using the syntax `\i`, where *i* is the group number. Suppose you wanted to match a date in the format YYYY-MM-DD, but allow any of several characters to be used in place of the hyphen as long as the same character is used in both places. Here's one way to work this out:

```
'\d\d\d\d-\d\d-\d\d'                 # brute-force match using hyphen
'\d{4}-\d{2}-\d{2}'                  # clearer
1 '\d{4}[-.,:']\d{2}[-.,:']\d{2}'    # various punctuation, can be different
2 '\d{4}([-.,:'])\d{2}\1\d{2}'       # punctuation must be the same
```

Remember that when including a hyphen in a character set, it must be the first character; otherwise, it is interpreted as indicating a range.

The last step in the preceding sequence enclosed the first character set in parentheses. This allowed a later "back reference" to whatever the set matched in the target, designated by `\1`.

Groups have a much more important use than back references, though: after a match has succeeded, the characters of the target that correspond to each group can be obtained from the match object returned by functions and methods that perform matches. This is a very powerful capability, which we'll explore as we go along—not just in the rest of this chapter, but as we use regular expressions in subsequent chapters as well.

The character | between two regular expressions is similar to `or` in Boolean expressions. It says to try a match with the first expression and then, if it fails, try the second. Any number of regular expressions can be separated by vertical bars. Each one is tried in turn until one matches or there are none left to try. If you enclose the whole sequence inside parentheses, later parts of the same regular expression and the resulting match object can tell what part of the target matched that group without even knowing which of the alternatives was matched.

The Actions of the re Module

The `re` module provides functions and methods that use regular expressions for a variety of actions. Matching and searching functions and methods return *match objects* when successful; otherwise, they return `None`. We'll discuss match objects a bit later. For now, we can just treat them as true values.

Functions

First, we'll look at some of the `re` module's functions (the meaning of the ***`flags`*** arguments to these functions is described in the next section):

`re.match(`*`pattern, target`*`[, `*`flags`*`])`
: Returns a match object if the regular expression ***`pattern`*** matches zero or more characters starting at the beginning of the ***`target`*** string, under the control of the ***`flags`*** value

`re.search(`*`pattern, target`*`[, `*`flags`*`])`
: Returns a match object for the first place in the ***`target`*** string that ***`pattern`*** matches

`re.findall(`*`pattern, target`*`[, `*`flags`*`])`
: Returns a list of all nonoverlapping matches in ***`target`*** as a list of strings or, if the pattern included groups, a list of lists of strings

`re.finditer(`*`pattern`*`, `*`target`*`[, `*`flags`*`=0])`
: Returns an iterator that produces a match object for each place in ***target*** that ***pattern*** matches

`re.split(`*`pattern`*`, `*`target`*`[, `*`maxsplit`*`=0[, `*`flags`*`=0]])`
: Returns a list of strings obtained by splitting the target at each place where ***pattern*** matches, up to ***maxsplit*** splits (all if `0` or omitted)

`re.sub(`*`pattern`*`, `*`replacement`*`, `*`target`*`[, `*`count`*`=0[, `*`flags`*`=0]])`
: Returns the string obtained by replacing each match of ***pattern*** to ***target*** with ***replacement***, up to ***count*** matches (all if `0` or omitted)

`re.subn(`*`pattern`*`, `*`replacement`*`, `*`target`*`[, `*`count`*`=0[, `*`flags`*`=0]])`
: Performs the same actions as `re.sub`, but returns a tuple `(`***`newstring`***`, `***`number`***`)`, where ***newstring*** is the string with the replacements made and ***number*** is the number of substitutions made

The module also provides two important functions that don't do matches or searches:

`re.escape(`*`string`*`)`
: Returns a string with all characters other than letters and digits preceded by slashes, for when you want to use a string as a literal regular expression pattern and don't want any characters taking on special meanings

`re.compile(`*`pattern`*`[, `*`flags`*`])`
: Returns a *regular expression object* constructed from ***pattern*** and ***flags***; the significance of this function will be discussed shortly

Flags

Some details of matching and searching are controlled by flags. Table 7-9 describes the `re` module's flag values. You can refer to a flag by either its short or its long form.

Table 7-9. Regular expression match and search flags

Long form	Short form	Effect
`re.ASCII`	`re.A`	Restrict `\w`, `\W`, `\b`, `\B`, `\s`, and `\S` to matching based on ASCII characters only, not the full Unicode character set
`re.IGNORECASE`	`re.I`	Ignore case
`re.LOCALE`	`re.L`	Make `\w`, `\W`, `\b`, `\B`, `\s`, and `\S` dependent on the local language
`re.DOTALL`	`re.S`	Allow `.` (dot) to match a newline
`re.MULTILINE`	`re.M`	Allow `^` to match at the beginning of each line in addition to the beginning of the string and `$` to match at the end of each line (after the newline) in addition to the end of the string
`re.VERBOSE`	`re.X`	Permit writing of formatted regular expressions that can even include comments

Functions in the module that take flag arguments expect a single value, not an arbitrary number of flags or a sequence of flags. When you need to specify multiple flags, you must use the | operator to combine them. The | operator—in an algebraic context, not within a regular expression—performs a *bitwise logical or*. Instead of treating the operands as two values that are each either true or false, bitwise logical operations treat each *bit* of each operand as a separate true or false value. For example, the bitwise or of 12 and 6 is 14:

```
    12    1100
|    6    0110
=   14    1110
```

It is a common computing technique to define a set of flags that are each integers with just a single bit. That allows the bitwise or operator to act as a set union operator for the flags, so that any combination of flags can be passed as a single value. For example, `re.I` is `2` (binary `00010`), `re.M` is `8` (binary `01000`), and `RE.S` is `16` (binary `10000`), so if you wanted to pass the combination of the three flags as an argument you would pass the expression `re.I | re.M | re.S`. (The order doesn't matter.) The binary result of that expression is `11010`, but using the names for the flags and the | operator allows you to ignore all that.

You probably won't use the `LOCALE` flag much. The `ASCII` flag introduces a limitation that is usually not necessary, except perhaps to avoid confusion in your mind about which Unicode characters are "space" or "word" characters. (This is an intricate topic that you'll probably want to stay away from if you can.) The `IGNORECASE` flag, as it suggests, ignores the case of letters in the target when performing the match.

You will use `DOTALL` and `MULTILINE` quite frequently, especially when using regular expressions to process sequence data. Without `DOTALL`, periods won't match newlines. Without `MULTILINE`, `^` and `$` will match only at the beginning and end of the target, respectively. When matching against a target string with internal newline characters, such as a sequence read directly from a FASTA file, you need these flags (specified as `re.DOTALL | re.MULTILINE` or `re.S | re.M`). Matching text containing URLs raises the same problem—you can write a regular expression to match hyperlinks (and we will), but there's no requirement in HTML that they appear entirely on one line.

The `VERBOSE` flag is quite different from the others. It doesn't control specific details of the match, but rather lets you write regular expressions in a more readable format, much like regular code. Unless preceded by a backslash, whitespace and comment characters are ignored.

Example 7-2 shows a regular expression that could be used to match decimal numbers. Note that the space after the `d` on the first line is ignored: *all* whitespace is ignored unless preceded by a backslash. Without the `VERBOSE` flag set, and assuming therefore that the comments and newlines are removed, the beginning of the pattern would read "one digit followed by one or more spaces" instead of "one or more digits."

Example 7-2. A VERBOSE regular expression for decimal numbers

```
r"""\d +  # match one or more digits (remember to use raw strings!)
    \.    # match a decimal point (not any character, because backslashed)
    \d *  # match 0 or more digits"""
```

Methods

The `re.compile` function returns a regular expression object that encapsulates a data structure built from the pattern and controlled by the flags provided as arguments. Regular expression objects have methods corresponding to the module's `match` and `search` functions. Some of them have ***startpos*** and ***endpos*** arguments that provide more control over matches and searches than the corresponding module functions. Using the compiled version is more efficient when the regular expression is used more than once, since otherwise the regular expression will have to be compiled each time it is used.

The parameters of match object methods are different from those of the corresponding module functions. Since the pattern and flags are incorporated into the regular expression object, they aren't included in the parameters. Compiled regular expression objects have the following methods:

`match(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns a match object if the regular expression matches zero or more characters starting at the beginning of ***target***

`search(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns a match object for the first match in the ***target*** string

`findall(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns a list of all nonoverlapping matches in ***target*** as a list of strings or, if the pattern included groups, a list of lists of strings

`finditer(`*`target`*`[, `*`startpos`*`[, `*`endpos`*`]])`
: Returns an iterator that produces a match object for each match

`split(`*`target`*`[, `*`maxsplit`*`=0])`
: Returns a list of strings obtained by splitting the target at each match, up to ***maxsplit*** splits (all if `0` or omitted)

`sub(`*`replacement`*`, `*`target`*`[, `*`count`*`=0])`
: Returns the string obtained by replacing each match with ***replacement***, up to ***count*** matches (all if `0` or omitted); ***replacement*** can be a function, in which case it is called with the result of each match and its return value is used to replace the corresponding match in ***target***

`subn(`*`replacement`*`, `*`target`*`[, `*`count`*`=0])`
: Same as `re.sub`, but returns not just the string, but a tuple (***string***, ***number***) where ***number*** is the number of matches; ***replacement*** can be a function, in which case it

is called with the result of each match and its return value is used to replace the corresponding match in *target*

Results of re Functions and Methods

Many of the `re` module's functions and regular expression object methods return match objects. These provide a wealth of information about the successful matches, some accessible as fields and some by calling methods.

The most powerful feature of match objects is that they capture the parts of each match in a target string that corresponds to one of the pattern's parenthesized groups. As a result, regular expressions and the functions and methods that use them can give much more than a yes or no answer to matches and searches. The groups captured in the match objects are an analysis of the structure of the matched string. We'll see how this works in examples later in this chapter.

Match Object Fields

Match object fields are accessed directly using dot notation. That is, if ***mobj*** is a match object, the `startpos` field will be accessed as ***mobj***`.startpos`. The fields include:

`startpos`
: The value of `startpos` passed to the function or method that returned the match object

`endpos`
: The value of `endpos` passed to the function or method that returned the match object

`re`
: The regular expression passed to the function or method that returned the match object

`string`
: The target passed to the function or method that returned the match object

`lastindex`
: The index (integer) of the last group in the match, or `None` if there were no groups

Match Object Methods

Match object methods include the following. In any match object method with group numbers as parameters, `0` refers to the entire match, `1` to the first parenthesized group in the match, `2` to the second, and so on:

`group([groupnumber1 [, groupnumber2 [, ....]]])`
: With no arguments, returns the entire match; with one argument returns the string that matched the corresponding group number, and with multiple arguments returns a tuple with the strings corresponding to the specified groups

`groups([default])`
: Returns a tuple containing all the groups of the match; for any group in the pattern that was not used in matching the target, the value in the tuple will be ***`default`*** (or `None` if not specified)

`start([groupnumber])`
: Returns the starting position of the target string that was matched by the group with index ***`groupnumber`***; with no arguments, returns the whole part of the target that was matched

`end([groupnumber])`
: Returns the ending position of the target string that was matched by the group with index ***`groupnumber`***; with no arguments, returns the whole part of the target that was matched

`span([groupnumber])`
: Returns the pair (*`startpos, endpos`*) corresponding to the range of the target string that was matched by the group with index ***`groupnumber`***; with no arguments, returns the first and last position of the whole part of the target that was matched

`expand(string)`
: Performs "backslash substitution" like the regular expression object `sub` method, but also replaces each occurrence of `\1`, `\2`, etc., in ***`string`*** with the value of the corresponding matched group

Putting It All Together: Examples

Now that we've looked at the features of the `re` module in some detail, it's time to put them together and see how they work in some useful examples.

Some Quick Examples

We'll start with a few simple examples to give you a more concrete idea of how to use the `re` module's facilities; then we'll move on to more substantial examples.

Using regular expressions to ignore case

The `find` and `index` methods of `str` do not provide any way to search while ignoring letter case. In some of our examples, we got around that problem by converting the target string to all uppercase or all lowercase. This is fine for simple cases, but it's awkward for situations in which you want to obtain information from the target string in its original case: you'd have to first search a single-case copy for the positions where

what you are looking for are found, then use those positions to extract the corresponding substrings from the original.

Regular expression searching provides a simple solution: just use the `re.IGNORE` flag. Despite their general power, there is no reason to reserve regular expressions for complicated situations. If you want to search `target` for `string`, ignoring case, this expression is all you need:

```
re.search(string, target, re.IGNORE)
```

Listing files in a directory, excluding some

Chapter 6 showed a function for listing the contents of a directory while excluding files with certain extensions. Example 7-3 shows a more flexible version of this function based on a list of file patterns instead of just file extensions. The function joins regular expressions in the list into a single pattern with the disjunction character | separating them.

Example 7-3. Directory listing filtered by regular expressions

```
def ls(path = ".", ignorepats = (r'.+\.pyc$', r'.+~$', r'^\#')):
    if ignorepats:
        pat = re.compile('|'.join(ignorepats))
        # construct an RE disjunction containing each item of
        # ignorepats to create an RE that matches any of the items
    for filnam in os.listdir(path):
        if not ignorepats or not pat.search(filnam):
            print(filnam)
```

Finding open reading frames

Regular expressions are ideal for all sorts of sequence feature detection. For example, they dramatically simplify finding open reading frames when compared to the kind of loops we saw in the examples of Chapter 4. We don't even have to worry about an extra base or two left over at the end of the sequence, the way we did in the loop-based code. Here's all we need:

```
openpat = re.compile('''
    ([TCAG]{3})*?                # 0 or more codons
    (ATG                         # start codon; begin match group
     ([TCAG]{3})*?               # 0 or more codons
     )                           # end match group
    (TAA|TGA|TAG)                # a stop codon
''', re.I | re.X )
```

The two occurrences of `([TCAG]{3})*?` force the match to consider three bases at a time. The pattern `[TCAG]{3}` is equivalent to `[TCAG][TCAG][TCAG]`. The first occurrence consumes codons that precede the `ATG` that begins the open reading frame, three at a time. The second finds the codons within the open reading frame. The parentheses around the first one are there not because we care about the result, but simply to make the

*? apply to the codon characters, since regular expressions don't allow one repetition indicator to directly follow another.

It would be nice if we could just use findall to find all the matches, but that would find the ATGs in *any* reading frame—for the same sequence, it might find one in reading frame 1 and another in reading frame 3, which makes no sense. We need to specify the reading frame we want the regular expression to use, so instead of findall we'll use match, which will force the regular expression to always start its match at the beginning of the string. We still need a loop to start each match at the end of the previous one and to collect the results, though. Example 7-4 shows a function that uses the compiled regular expression to find all the open reading frames starting in a given frame.

Example 7-4. Finding open reading frames with a regular expression

```
def open_reading_frames(seq, frame=1):
    lst = []
    matchobj = openpat.match(seq, frame-1)
    while m:
        lst.append(m.group(2))
        matchobj = openpat.match(seq, matchobj.end())
    return lst
```

An even more compact version can be written as a generator, as shown in Example 7-5.

Example 7-5. Regular-expression-based open reading frame generator

```
def open_reading_frames_generator(seq, frame=1):
    matchobj = openpat.match(seq, frame-1)
    while matchobj:
        yield matchobj.group(2)
        m = openpat.match(seq, matchobj.end())
```

The expression open_reading_frames_generator(seq, frame) will return a generator, and every time next is called on that generator the next open reading frame will be returned.

Extracting Descriptions from Sequence Files

The web page at *http://rebase.neb.com/rebase/rebase.seqs.html* has links for viewing and downloading DNA and protein sequences for restriction enzymes. The sequences are grouped into files according to enzyme category, but the format of the files is consistent. Assume we've downloaded one of the DNA files to the file *rebaseseqs.txt*, and we want to extract the description data for each of its sequences.

The description data in these files has a rigid format:

```
>EnzymeName   RecognitionSequence  SequenceLength Type
```

The number of spaces separating the fields is fixed: 3, 2, and 1. Not all sequences in these files designate a RecognitionSequence field. Examples in Chapters 3 and 4 showed a variety of ways of approaching a problem like this using comprehensions, loops, and

iterations. We're going to see now how much simpler a solution using regular expressions can be. If we can define an appropriate regular expression, we can use `re.findall` to find all the descriptions, doing the repetitions for us that we would otherwise have to program with a comprehension, loop, or iteration. Not only that, but a regular expression with grouping will even parse the results into their fields.

The first step is to create a regular expression that matches the whole description line. To simplify things, we'll assume that the contents of the datafile have been read and named `data`:

```
re.findall(r'^>.*$', data, re.MULTILINE)
```

This regular expression matches target substrings characterized by:

- `>` at the beginning of a line
- Followed by any characters
- Until the end of the line

We use `re.MULTILINE` so `^` and `$` can match lines within `data`; otherwise, they would match only at the beginning and end of the whole string. Using this pattern with `re.search` returns a list of strings such as:

```
'>M.AacDam   GATC  855 nt'
```

The next step is to match the fields of the data individually.

```
re.findall(r'^>[^ ]+   [\w]*  \d+ .+$', data, re.M)
```

When you are first learning about them, even a regular expression as basic as this one looks quite cryptic. When you encounter one, try writing down what you think it means. Just go from left to right, as you would with an algebraic expression. This one is read as;

- A `>` at the beginning of a line
- One or more nonspace characters, captured as a group
- Exactly three spaces
- Zero or more word characters, captured as a group
- Exactly two spaces
- One or more digits, captured as a group
- Exactly one space
- One or more characters at the end of the line, in a group
- The end of the line

You can also use the `re.VERBOSE` (`re.X`) flag and comment the expression itself. Just be sure to put backslashes in front of any spaces inside the regular expression, because with that flag *all* whitespace—space, tabs, newlines—is ignored, apart from whitespace

indicated by escaped characters (`\ \n\t`). Here's how we could combine our regular expression with the explanation that follows it:

```
    r'''
^>          # a > at the beginning of the line
([^\ ]+)    # one or more nonspace characters, captured as a group
\ \ \       # exactly three spaces
([\w]*)     # zero or more word characters, captured as a group
\ \         # exactly two spaces
(\d+)       # one or more digits, captured as a group
\           # exactly one space
(.+?)$      # one or more characters at the end of the line, in a group
'''
```

The result returned by **`findall`** is the same as for the earlier regular expression. We take this step in developing the regular expression to make sure that we have the pieces described accurately. For instance, if there were only two spaces after the first **`+`**, there wouldn't be any matches. In the next step we simply put parentheses around the parts we are interested in so that we can extract them from the match objects as groups:

```
re.findall(r'^>([^ ]+)   ([\w]*)  (\d+) (.+$)', data, re.M)
```

The result we get now is different, and more useful: instead of a list of strings, we get back a list of tuples of strings. Each tuple contains the portion of the matched string corresponding to one of the parenthesized portions of the regular expression. For the line:

```
'>M.AacDam   GATC  855 nt'
```

the tuple in the list of results would be:

```
('M.AacDam', 'GATC', '855', 'nt')
```

Earlier in the chapter, it was observed that regular expressions used with bioinformatics data usually contain nongreedy repetition characters. The reason we didn't need them here is that the pattern *anchored* the match to the beginning and end of a single line. Since the goal was to extract single lines and the information they contained, using greedy repetition characters did not create any problems in this case.

Extracting Entries From Sequence Files

Also in Chapters 3 and 4 were a number of examples that involved reading descriptions and/or sequences from FASTA files. They required function definitions to use string methods to search and split either individual lines or the entire file contents. As a result they were somewhat complex.

Using regular expressions to extract entries

Regular expressions make getting the next sequence from the file a much simpler task. To extract a description and all the sequence characters, the following suffices:

```
re.search(r'^>[^>]*', src).group()
```

This reads as "match the next `>` that starts a line through all characters that aren't a `>`." This takes care of the problem of needing to read the description line to find the end of the previous sequence. We'd like to do better than that, though. First we'll separate the description from the sequence, capturing two groups:

```
re.search(r'^>(.*)$([^>]+)', src, re.M).groups()
```

The plus at the end prevents the pattern from matching a (possibly partial) description line not followed by any sequence characters, which could happen at the end of the `src` string. The `$` serves to split the match into everything on the description line and everything else.

Then, all we have to do is use ordinary string functions to clean up each of the two groups returned. Example 7-6 shows a complete definition of `next_item` that doesn't use subfunctions to read the fields from the file. (The cleanup code, however, *is* in a separate function, so those details are out of the way of the basic matching loop.) In this example, `next_item` searches for the next part of the text that matches the compiled regular expression, then calls `extract_from_match` to process the result. If there is no match, `matchobj` will be `None` and `extract_from_match` will return `None`. Otherwise, it will return a tuple of the form (*`description`*, *`sequence`*). The description, found in group 1 of the match object, is broken up into a tuple by splitting at the occurrences of `|` and stripping the spaces from the results. The sequence, found in group 2, has its newline characters removed.

Example 7-6. Defining next_item for FASTA with a regular expression

```
pat = re.compile(r'^>(.*)$([^>]+)', re.MULTILINE)

def next_item(src):
    return extract_from_match(pat.search(src))

def extract_from_match(matchobj):
    return (matchobj and                              # return None when match fails
            (([field.strip() for field in matchobj.group(1).split('|')],
              matchobj.group(2).replace('\n', '')),
              matchobj.end()))
```

A really powerful feature of this solution is that we can accommodate variations in the file format with only slight changes to either the pattern or the cleanup in `match_item`.

Keeping track of the position between calls

Unfortunately, this definition of `next_item` has a major problem. Regular expression operations expect their arguments to be strings (or things like strings, such as bytes and bytearrays); they do not work with open files. The `src` parameter of `next_item` must therefore be a string representing the entire file contents. However, the preceding definition will return the same item each time it is called.

We can fix this problem by keeping track of where the last match ended and starting from there the next time `next_item` is called. This is called *state maintenance*. One way to keep track of a value in between function calls is:

1. Add a parameter for the previous value to the function.
2. Return the value along with the other value(s) the function returns.
3. Have each function supply the previous value as one of its arguments.
4. Assign a name to the value part of the result of the function.

This is probably easier to illustrate with an example than with a written description. Let's consider how we could implement a `findall` function for plain strings using `str.find`. Example 7-7 shows a straightforward iterative definition like many we've already seen. Each time the target substring is found, the function adds its position to a list it is accumulating and then searches again, starting at the next character after the substring.

Example 7-7. A findall function for ordinary strings

```
def findall(string, substring):
    """Return a list of all nonoverlapping positions in string where substring appears"""
    positions = []
    pos = string.find(substring)
    while pos >= 0:
        positions.append(pos)
        pos = string.find(substring, pos + len(substring))
    return positions
```

Example 7-8 shows an alternative approach with two functions: one just finds the next occurrence of the target, and the other calls it and collects the results. There's no particular advantage to doing things this way for this simple example; it simply illustrates the technique so we can use it for something more substantial next.

Example 7-8. Keeping track of a value between calls

```
def findnext(string, substring, startpos=0):
    """Return the position of the next nonoverlapping occurrence of
    substring in string, beginning at startpos where substring appears"""
    pos = string.find(substring, startpos)
    if pos < 0:
        return -1, -1
    else:
        return pos, pos + len(substring)

def findeach(string, substring):
    positions = []
    pos, startpos = findnext(string, substring)
    while pos >= 0:
        positions.append(pos)
        pos, startpos = findnext(string, substring, startpos)
    return positions
```

The important point to note about these two functions is that `findnext` needs to know where to start looking for the substring each time it is called. It's up to `findeach` to keep track of that next starting position between calls. We've defined `findnext` so that it returns two positions: the position where the substring starts and the position where search should recommence. In `findeach`, the two values returned from `findnext` are assigned, and the second is fed back into `findnext` when it is called. This example is so simple that `findeach` could compute where to start the next search as easily as `findnext` could, but if `findnext` does anything more interesting that might well not be the case.

Example 7-9 shows a definition of `next_item` that applies this technique to keep track of the search position between calls. It also shows an example of the function's use.

Example 7-9. Defining next_item to return and receive a position

```
def next_item(src, pos):
    return extract_from_match(pat.search(src, pos))

def extract_from_match(matchobj):
    return ((None, -1)                                  # pos doesn't matter here
            if not matchobj
            else
            (([field.strip() for field in matchobj.group(1).split('|')],
              matchobj.group(2).replace('\n', '')),
             matchobj.end()))                           # new value of pos to be remembered

def find_item(src, testfn):
    item, pos = next_item(src, 0)
    while (item):
        if testfn(item):
            return item
        item, pos = next_item(src, pos)
```

Another thing we can do is wrap `next_item` in a generator. Keeping track of the values of parameters and other names assigned in the function is one of the primary responsibilities of the generator mechanism. Another is keeping track of where the `yield` occurred in the definition (there may be several) so that execution can resume right after that statement when `next` is called on the generator object. Example 7-10 shows what that would look like.

Example 7-10. Defining next_item as a generator

```
def item_generator(src):
    pos = 0
    item, pos = next_item(src, pos)
    while item:
        yield item
        item, pos = next_item(src, pos)

def find_item(src, testfn):
    itemgen = item_generator(src)
    item = next(itemgen)
```

```
    while (item):
        if testfn(item):
            return item
        item = next(itemgen)
```

We've chosen to leave the definition of `next_item` unchanged from its definition in Example 7-9. Instead of adding complexity to `next_item` and `extract_from_match`, we've defined a second function to produce the generator object that keeps track of the position. Each time `next` is called on the generator object returned by `item_generator`, execution resumes in `item_generator` and continues until the `yield`. All this generator is doing is holding onto the value of `pos` between calls to `next_item`.

Notice also the changes in `find_item`. It no longer does anything to keep track of the position and provide it as an argument to `next_item`. In fact, it no longer calls `next_item` at all. Instead, it calls `item_generator` once at the beginning of the function, then uses `next(`*`itemgen`*`)` each time it wants another item.

> Example 5-4 showed a class whose methods share state using instance fields. The problem here is not quite the same as the one solved by the class example. There, the emphasis was on avoiding a lot of useless argument-passing for values that were used unchanged by many methods. Here, the challenge is to *maintain* changing state between one function call and the next, not just *share* it. In the technique demonstrated in the preceding example, one function returns values to its caller that it will need again later, and the caller passes them back as arguments the next time it calls the function. State maintenance is even more of a reason to use a class than state sharing, and usually you would use a class rather than the tricks of the previous example. We'll see examples of a two such classes in Chapter 8 (in Examples 8-17 and 8-18).

Buffering input

Another problem with these definitions is that code using them must read the entire contents of a file to pass `find_item`. If the file is extremely large this could take an inconvenient amount of time and space, especially if the sequence is located early in the data. And even if the intent is to do something with every sequence in a file, you still may not want to read the entire contents all at once.

The solution to this sort of problem is to interpose a *buffer* between the file and `next_item`. A buffer is a standard programming mechanism that turns processing a source or destination—reading from a file, writing to a file, searching a string, etc.—into a two-step process. A file buffer contains a moderate portion of the file's bytes or characters. Reading is done from the buffer, rather than from the file itself. When the end of the buffer is reached, its contents (or most of them, anyway) are discarded and another piece of the file is read into the buffer. For writing, the process is reversed: writing is directed to the buffer, and when it's full its contents are written out to the file and the buffer is emptied. (This is called *flushing* the buffer.)

The main reason for buffering is that input and output operations that involve hardware storage devices are many orders of magnitude slower than reading from and writing into the computer's memory. Buffering replaces many small external interactions with many small internal interactions and only the occasional bulk transfer from or to external storage. As noted in Chapter 6 (in "The Python runtime environment: sys" on page 213), `sys.stdout` is a buffered file stream for this reason.

Python's implementation of input and output streams includes buffering capabilities, but programs can also implement their own, higher-level buffering mechanisms. We can add a buffer to Example 7-10's definition of `item_gen`, enabling it to pass strings to `re` functions and methods without first having to read the entire contents of a huge file into memory. This will improve efficiency in the event that a call to `find_item` locates a targeted item after examining only a small portion of the file.

The strategy will be to read a number of characters from the file and read sequences from the string returned; then, when all the sequences of the string have been read, another chunk of characters will be read. What's tricky about this is that the string beginning with the last `>` is almost definitely incomplete; the rest of the sequence will be in the next chunk.

The process is illustrated in Figures 7-1, 7-2, and 7-3. It's not easy to develop code like this, but it shouldn't be too hard to understand.

Example 7-11 shows a `get_item` function added to manage the buffering.

Example 7-11. Buffered regular expression search of a file

```
def get_item(fil, buffer, pos, chunksize):
    """Return a possibly incomplete item along with the new value
    of buffer and the end position of the successful match"""
    item, endpos = next_item(buffer, pos)          # initialize loop
    while not item:                                # look for next item
        chunk = fil.read(chunksize)                # read next chunk
        if not chunk:
            return None                            # end of file
        buffer += chunk                            # add chunk to buffer
        item, endpos = next_item(buffer, pos)      # try again
    return item, buffer, endpos                    # beginning of an item
```

The purpose of `get_item` is to call `next_item` until it returns an item. Each time `next_item` fails to find an item, `get_item` reads `chunksize` characters from the file and appends them to the buffer. When enough of the file has been read into the buffer for `next_item` to find an item, `get_item` returns the item, the (possibly extended) buffer, and the ending position of the match. This follows the approach discussed earlier, whereby values are maintained between calls by returning them to the caller and having the caller pass them back in. In this case, two values are tracked: the buffer itself and the position within the buffer where the search should resume.

The end of the buffer might contain a FASTA description and the beginning of its sequence, but not the entire sequence. How can we know whether `get_item` has

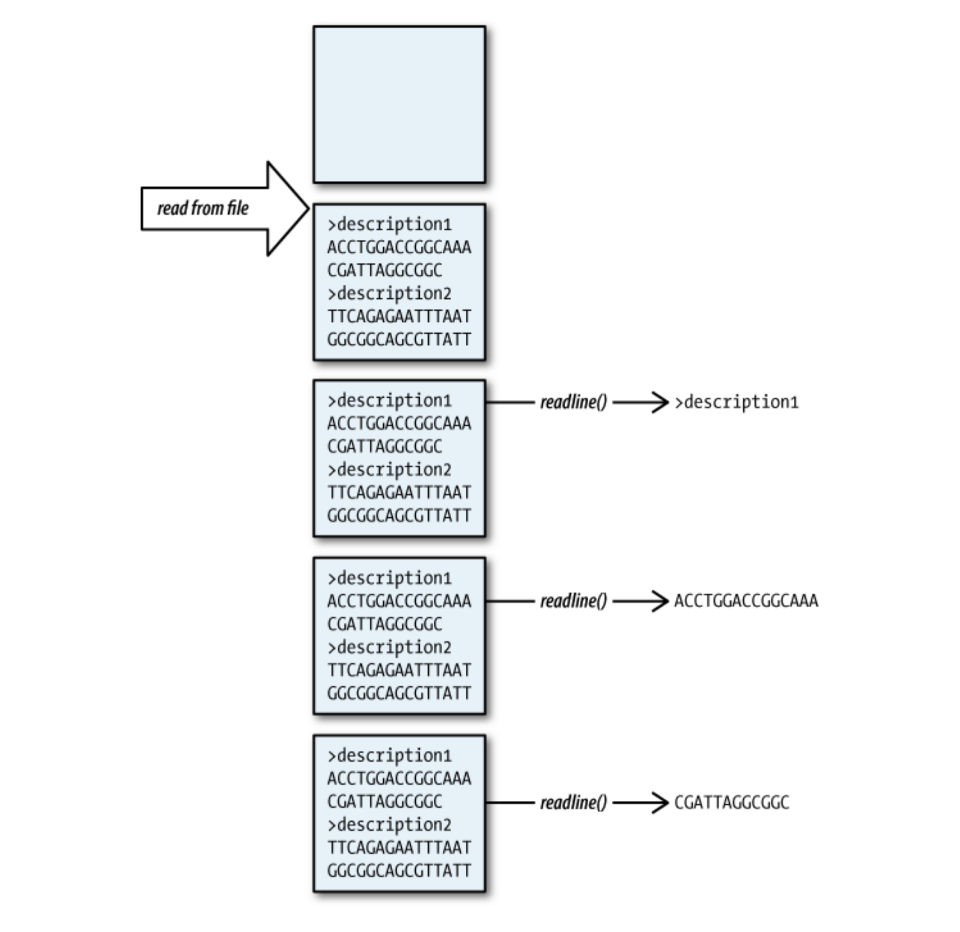


Figure 7-1. Buffering input while doing searches, part 1

returned a complete sequence? The solution is to always read a second sequence ahead of the one to be returned. As long as the beginning of another sequence is found, we know the current one is complete. When the current sequence is the last in the buffer, looking for the next one will cause the buffer to be extended until a next sequence is found. At that point, the current sequence must be reread to get its full contents.

The outline of the code is similar to a lot of loop-based functions: it reads an item, then loops until there are no more items. In this case, however, the code starts by reading *two* items and loops until there is no more *second* item. The technical term for reading one item beyond the targeted one is *lookahead*, for obvious reasons. Lookahead is required whenever some information from the next item is needed to process the current item (including, at a minimum, whether there *is* a next item). In the normal case, each time around the loop, the name of the first item is rebound to the second item and a new second item is read. This code also has to deal with the abnormal case when no

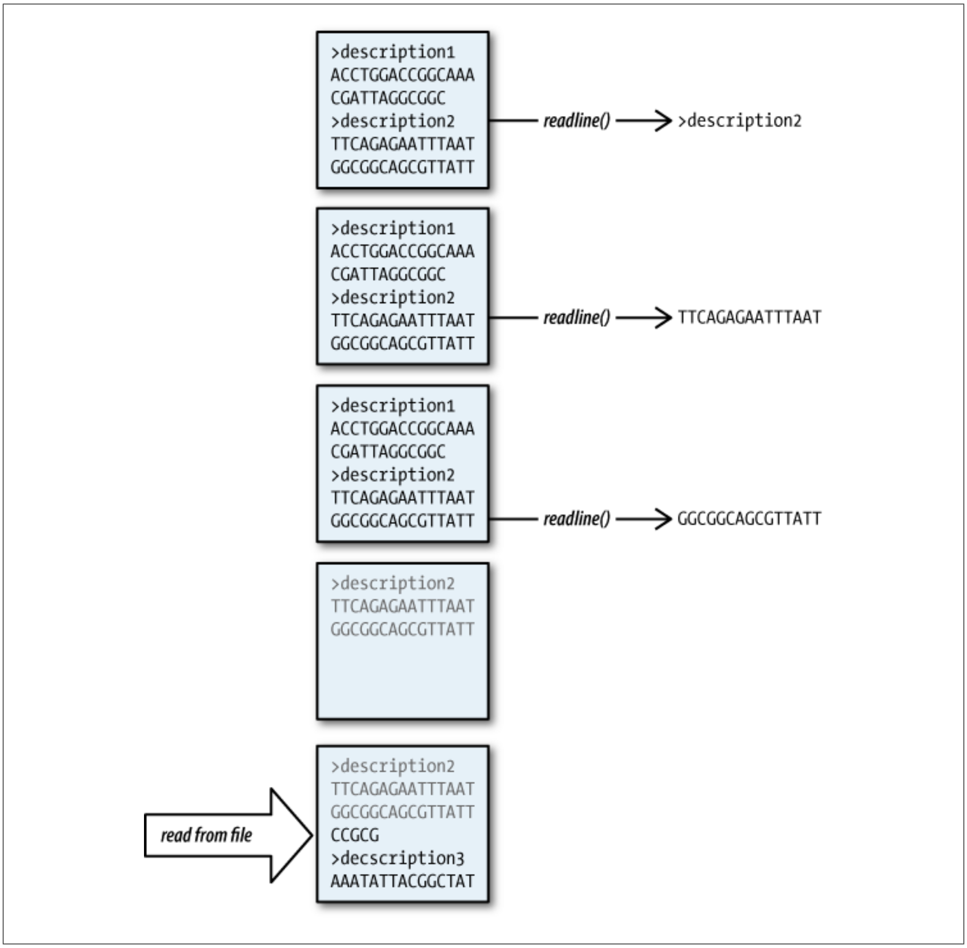


Figure 7-2. Buffering input while doing searches, part 2

second item is found, because that is when the buffer will be extended (except at the end of the file). The outline looks like this:

```
initialize
first item = read item
nextitem = read item
while nextitem:
    if finding the next item caused the buffer to be enlarged
        reread item from the buffer because it was incomplete
        shorten buffer to exclude previously read items
    yield item
    item = nextitem
    nextitem = read item
yield item        # at end of file
```

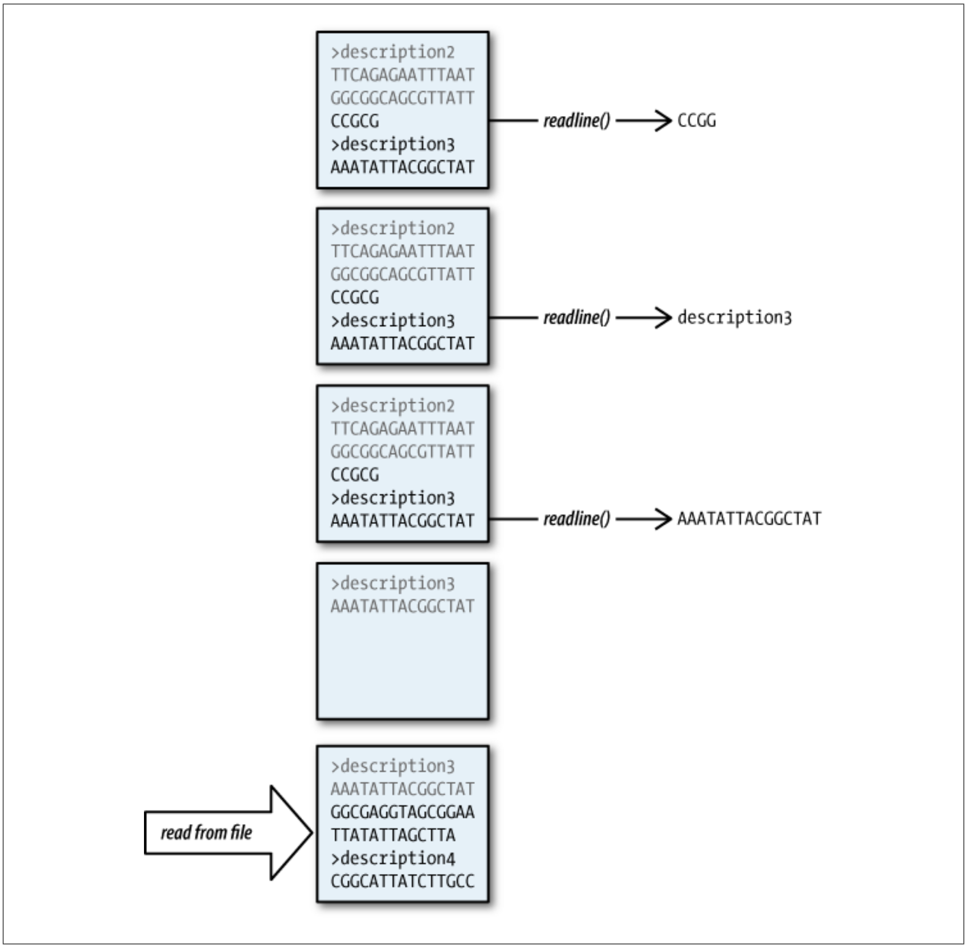


Figure 7-3. Buffering input while doing searches, part 3

Example 7-12 shows code that implements this outline. The solution is complicated by having to maintain three positions: the beginning of the current item (`curpos`), the beginning of the next item (`nxtpos`), and the end of the next item (`nxtend`). (At the end of the buffer, `nxtend` will equal the length of the buffer). A numbered line in the code, together with any following unnumbered lines, implements the correspondingly numbered part of the outline.

Example 7-12. Buffered regular expression generator for a file

```
def item_generator(file, chunksize=4000):
1     curpos = 0
2     itm, buffer, nxtpos = get_item(file, '', curpos, chunksize)
3     nxtitm, buffer, nxtend = get_item(file, buffer, nxtpos,
```

```
    while nxtitm:
        if len(buffer) > nxtend:
            itm, nxtpos = next_item(buffer, curpos)

            nxtend -= nxtpos

        yield itm
        itm, curpos, nxtpos = nxtitm, nxtpos, nxtend
         nxtitm, buffer, nxtend = get_item(file, buffer, nxtpos,

     yield next_item(buffer, pos)[0]
```

Tips, Traps, and Tracebacks

Tips

The importance of constructing regular expressions piece by piece cannot be emphasized enough. Make sure each pattern is working as expected before making it more complex. Does this sound familiar? It should: this book has repeatedly stressed this approach for defining functions. Unfortunately, regular expressions don't have the ability to call other regular expressions, so you can't really build them out of truly separate pieces. You can, however, still proceed in a step-by-step manner, increasing the complexity as you go.

Debugging regular expressions is far more difficult than debugging programs. You can't work your way step by step through the match the way you can with a Python debugger. There are tools for testing and debugging regular expressions, but even without one you can often accomplish a lot by simply printing the results of a `findall`, one to a line. Another trick that's useful when a regular expression isn't doing what you expect is to remove parts of it gradually and repeat the search until you locate the problem.

As you start building more complex regular expressions, you will experience more than a little frustration. Here are a few pointers:

- Use a regular expression testing tool such as re-try. You don't need the entire source file for testing your regular expression—just grab a section that will produce a match or two once your regular expression is working.
- Remember to use raw strings for regular expressions.
- In general, it's probably not worth trying to tune a regular expression so that a group such as FASTA sequence characters matches multiple lines but excludes the newlines. Just use `replace('\n', '')` on the result.
- Separating fields by a delimiter character such as a vertical bar on a FASTA description line can be done within the regular expression, but if you are having trouble accomplishing that, just have the expression match the whole line and use the following to get a list of the fields:

```
[field.strip() for field in line.split('|')]
```

- To search any of a list of patterns, search for:

```
'|'.join(list_of_patterns)
```

Traps

- Remember to use the | operator to combine flags; `re` functions that take a flag argument, including `re.compile`, only accept one such argument. If you forget and use commas instead you will get an error message like the following:

```
TypeError: fn() takes at most 2 positional arguments (3 given)
```

  The exact numbers will depend on which function you called and how many arguments you supplied.
- Beware of vertical bars in regular expressions! They represent disjunction. If you are trying to match a vertical bar, as in FASTA headers, you must use a backslash (\|).
- When attempting to capture a group, make sure the repetition characters are inside the group unless you really do mean to repeat the group. That is, if you want a sequence of digits, write `(\d+)` and not `(\d)+`. The latter will return the last digit only, a rather mysterious result. This is an easy mistake to make, especially if you casually put parentheses around a small part of a partially completed regular expression.
- When matching the contents between "delimiter" characters—quotes, parentheses, brackets, braces, angle brackets, etc.—a `*` or `+`, and even a nongreedy `*?` or `+?`, will often match too much of the target string. In these situations you should match a repetition of "anything but the closing delimiter." For example, to match something enclosed in quotes, use `"[^"]+"` instead of `".+?"`.
- Make sure to use the `re.DOTALL` (`re.S`) flag if a newline can occur within text you want matched by `.*`, `.+`, `.*?`, or `.+?`, and not to use it if you don't want newlines matched.
- Make sure to use `re.MULTILINE` (`re.M`) if you are using anchors—`^` for the beginning of a line and `$` for the end. Otherwise, `^` will match only the beginning of the entire string, and `$` will match only the end.
- If you use `re.VERBOSE` (`re.X`), make sure to put a backslash in front of any space that you mean to be part of the pattern; otherwise, it will be ignored just like the whitespace you use for making the string readable.
- A negated character set such as `[^>]*` will match line feeds. This pattern:

```
r'^>[^>]*'
```

  will match an entire FASTA entry—the description and all of the sequence characters, with newlines embedded, regardless of whether `re.DOTALL` is specified.

If `re.MULTILINE` is specified, a dollar sign added to the end of that expression will have no effect because the character set will still match newlines. To prevent negated character sets from matching newlines you must include the newlines explicitly in the character set, either as `\n` or `\s` (which would exclude all whitespace).

Tracebacks

- The same kinds of errors that occur when parentheses aren't balanced in code, especially in algebraic expressions, can easily happen with group parentheses in regular expressions, producing the error:

  ```
  sre_constants.error: unbalanced parenthesis
  ```

- The error:

  ```
  sre_constants.error: nothing to repeat
  ```

  can be particularly irksome. If you are developing a pattern incrementally, the obvious place to look for the source of this problem is where you last added a repetition character, or perhaps a pair of parentheses.

- The error:

  ```
  sre_constants.error: multiple repeat
  ```

  means that somewhere in the regular expression there are adjacent repetition characters. Perhaps the most likely culprit is putting the question mark before the plus sign or asterisk in a nongreedy repetition, instead of after. This problem can also arise when, while developing a regular expression, you accidentally remove the character(s) that separated two repetition characters.

- Unfortunately, the error won't point you to where in the expression the problem occurs; in fact, if you are using `re.VERBOSE` (`re.X`) it won't even show you which line to look at—the error always points to the end of the function call in which the expression appears. Note also that the line number of the offending function call appears early in the traceback, followed by many calls to `compile`, `parse`, and related subfunctions. This contrasts with most tracebacks, where it is the last few lines that show you where in your code the error occurred.

CHAPTER 8
Structured Text

In Chapter 6, we took a very brief look at the `csv` module that is used to read and write lines of tab- or comma-separated values, with each line corresponding to one item in the file. We've also looked at a variety of ways to scan files looking for certain patterns of data, including using `str` methods and regular expressions. Files that are in tab- or comma-separated values format, FASTA files, GenBank files, and many other file formats encountered in bioinformatics work are called *flat files*.* What is "flat" about them is that they are just text files: the data has no explicit structure beyond agreed-on conventions regarding special characters, blank lines, whitespace, etc. They can have introductory material before the data, other material after the data, several sets of data in one file, and so on.

The opposite of "flat" in this context is *structured*. A structured text file contains *elements*, each of which can have *attributes* and/or "sub" or *child* elements. There can be different kinds of elements, and in general there are rules specifying what attributes and children each kind of element can have. The linear approaches for processing text files that we've seen so far are inadequate for structured files, essentially because the files are two-dimensional. This chapter describes some ways to process structured files.

HTML

An obvious example of a structured file format is basic HTML. (We'll ignore all the fancy stuff like JavaScript, frames, and so on.) HTML's structuring rules are a bit loose, but we can ignore those details. The top-level structure of an HTML file is:

```
<html>
    <head>
        . . . tags and text . . .
    </head>
```

* In computer science the term "flat file" usually has a stricter meaning, referring only to text files with one item per line, each having fields designated by separators (commas, tabs, vertical bars, spaces, etc.) or conforming to some specified number of characters. Files in formats such as FASTA and GenBank would be considered "free form," even though they have some regularity.

```
        <body>
            . . . tags and text . . .
        </body>
    </html>
```

Each HTML element opens with a *start tag*, which begins with one of the tag names defined by the HTML specifications and is enclosed in angle brackets (<>). Tags may have attributes, in the form ***attribute-_name*** = ***attribute_value***. Elements may also have *content*—either text or other elements—that appears after the start tag. Tags with no content may be closed by a slash at the end of the start tag instead of an end tag: for instance, <hr/> to draw a horizontal line. Comments, which can span many lines and cause the text between them to be ignored by a browser displaying the HTML, begin with the characters <!-- and end with -->.

The end of a tag with content may be indicated by </***tagname***>, where ***tagname*** is the same as the name in the start tag, but for the most part the HTML standards don't require that. For instance, it is common practice to mark the beginning of a paragraph with a <p> tag but not to bother with the closing </p>, leaving the browser to figure out that the paragraph has been closed because a new one has been started (among other reasons).

> People often use the term "tag" to refer to the structure element a tag specifies, in addition to using it for the tag itself. In the rest of the book we will use just the tag name when talking generically about a tag, as in the phrase "an hr tag." When we specifically mean a start or end tag, we will use angle brackets.

Some tags in a file contain others. For instance, the top-level html tag contains the body tag, which contains the rest of the tags in the file. Many of those could be tags for headers or paragraphs whose content is just text. Other kinds of tags have "child" tags as their content. Lists are an example:

```
    <ol>
        <li>Item 1</li>
        <li>Item 2</li>
        <li><a href="http://www.example.org">Item 3</a></li>
    </ol>
```

This bit of HTML says to create a numbered list (ol) containing three items (each li).

There are other sorts of things inside HTML files, such as a few tags at the beginning that contain information about the file itself and embedded scripts that you usually won't want to consider as part of the document. There are also special notations such as *named entities* for punctuation, Unicode characters, and so on.† You'll see those most often where characters with meaning inside HTML itself are part of content; for

† You can find some good references at *http://www.w3schools.com/tags/ref_entities.asp*, *http://www.entitycode.com/#common-content*, and *http://www.escapecodes.info*.

example, `&` or `&` for ampersand and `<` or `<` for `<`. Each named entity begins with an ampersand, ends with a semicolon, and contains either letters or a pound sign followed by a base-10 number that refers to specific characters in standardized character sets.

Unicode Characters in HTML Files

HTML files often contain *actual* Unicode characters, not just named entities that *represent* them. Any attempt to read from a file that contains Unicode characters will cause an error unless you provide the optional keyword argument `encoding='utf-8'` in the call to `open`. Since the default encoding is a subset of Unicode, you can use that argument even with files that contain just ordinary ASCII characters.

If you want to use the file's actual encoding rather than guessing that `'utf-8'` will work, you can find it in the file itself. Any HTML file that contains non-ASCII characters should have a tag like the following in its `head` section:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

The following code will extract the encoding from an HTML file. It opens the file in binary mode to avoid any encoding issues, reads some characters, uses a binary regular expression to look for the tag that specifies the encoding, closes the file, and (if it finds one) returns the value of the tag's `encoding` attribute decoded into a string. Subsequent examples that read from HTML files will use `get_html_file_encoding` to determine the encoding argument they should provide when opening the file for processing:

```
encoding_pat = re.compile(
    b'<meta [^>]*?content *= *[^>]*' +          # just to fit line
    b'charset *= *([a-zA-Z0-9-]+)',
    re.I | re.A)

def get_html_encoding(html):
    encoding = encoding_pat.search(html)
    return encoding and encoding.group(1).decode()

def get_html_file_encoding(filename):
    with open(filename, 'rb') as file:
        return get_html_encoding(file.read(2000))
```

Simple HTML Processing

Some simple search or replace operations can be performed on HTML files using a text editor, a web design tool, Python, or another programming language. Suppose you find a web page with many interesting hyperlinks, and you want to extract them from the rest of the page. You could just search for `'<a '`, then for `'href='`, and so on, but there are too many possible variations to handle. You need regular expressions.

Searching HTML text

In this section we'll develop a program to extract all hyperlinks and associated text from a web page saved as an HTML file. The program will look a lot like some of the other examples we've already seen in the book. Example 8-1 shows the high-level functions that won't change as we develop the code further and the test at the end that lets us run it as a command-line program.

Example 8-1. Extracting <a> links from an HTML file, step 1

```
def print_atags_in_files(lst):
    for filename in lst:
        print(64*'-')                        # draw separator line
        print_atags_in_file(filename)

def print_atags_in_file(filename):
    with open(filename, encoding=get_html_file_encoding(filename)) as file:
        print_atags(file.read())

def print_atags(string):
    for atag in get_all_atags(string):           # defined below
        print_atag(atag)

def print_atag(results):
   print(textwrap.fill(results[0], 75),
         textwrap.fill(results[1], 75),
         sep='\n',
         end='\n\n')

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: show_atags htmlfile ...")
    else:
        print_atags_in_files(sys.argv[1:])
```

This time we've shown the definitions "top-down"—i.e., callers above callees. The order of the function definitions within a file doesn't matter to Python; it's up to you to decide how to organize the file for easiest understanding. Often that means putting the "high-level" functions at the top, so someone interested in using the program can see them without having to read through all the details. Then again, if you begin the program with a docstring describing the functions meant to be called from other files, there would be no need for a user to read the code to find them.

Next we'll present a series of definitions for `get_all_tags`, omitted from the preceding code. URL references span a wide range of formats. Before trying to formulate regular expressions to match them, it's well worth defining what we're trying to capture. As the program gets used it may produce some unexpected output, at which point we can

choose to accommodate the new example it reveals or make a note that we are ignoring the problem.

There are a variety of start tags that can contain `href` attributes, such as `<img>`, but we are only interested in `<a>` tags. If we discover that we are interested in others, we can always add them later. Even in trying to describe the URL pattern, we will follow the incremental approach exhibited in our earlier code examples. We'll start by capturing the `<a>` tags, their attributes, and their content and see what they look like before tackling more complex aspects of the patterns. That will actually bring us quite far toward our goal, but, as we'll see, may include a variety of extraneous information. Example 8-2 shows the initial regular expression.

Example 8-2. Extracting <a> links from an HTML file, step 2

```
atagpat = re.compile(
    r'''<a\s+(.*?)>     # the <a> tag attributes up to first >
        (.*?)           # capturing the entire content up to first </a>
        </a>            # the </a> end tag
     ''',
    re.IGNORECASE|re.MULTILINE | re.DOTALL | re.VERBOSE)

def get_all_atags(string):
    return atagpat.findall(string)
```

Here are some of the results extracted from the output of the program run on the results of an advanced search for the phrase "nonsense mediated decay" at *http://biology.plosjournals.org*. (The actual output would be several hundred lines long.) For each link the content is printed first, followed by the attributes of the `<a>` tag starting on a new line. Where more than two lines are shown it is because one or both of these lines overflowed the line length in the Python window where the code was executed:

```
PLoS.org
href="http://www.plos.org" title="PLoS.org"

Site Map
href="http://journals.plos.org/plosbiology/sitemap.php"
title="PLoS Biology Site Map"

Readers&#39; Responses
href="http://biology.plosjournals.org/perlserv/?request=read-response&#38;
issn=1545-7885&#38;past_days=360" title="For Readers: Readers&#39; Responses"

A Combinatorial Code for Splicing Silencing: UAGG and GGGG Motifs
href="http://biology.plosjournals.org/perlserv/?request=get-document&#38;
doi=10.1371/journal.pbio.0030158"

Posttranscriptional Gene Regulation by Spatial Rearrangement of the 3&#8242;
Untranslated Region
href="http://biology.plosjournals.org/perlserv/?request=get-document&#38;
doi=10.1371/journal.pbio.0060092"
```

Now that we have the attributes and the content, we want to do some cleanup. First, we'll extend the pattern to extract the value of the `href` attribute. This is a bit tricky—the equals sign can be surrounded by spaces and newline characters, and the value can be surrounded by a pair of single or double quotes. We'll allow for the quotes, but we won't worry about them matching, and quotes aren't allowed inside the value of the `href` attribute. We will also assume there is no whitespace within the value of `href`. The modified regular expression is:

```
r'''<a.*?href\s*=\s* # href=, optional spaces
    ['"]?            # optional quotes
    ([^'"\s]*)       # the value: no ', ", or whitespace
    ['"]?            # optional quotes
    [^>]*?>          # the rest of the tag
    (.*?)            # capturing the entire content
    </a>             # the </a> end tag
 '''
```

The results for the `<a>` tags shown previously are now:

```
PLoS.org
http://www.plos.org

Site Map
http://journals.plos.org/plosbiology/sitemap.php

Readers&#39; Responses
http://biology.plosjournals.org/perlserv/?request=read-response&#38;
issn=1545-7885&#38;past_days=360

A Combinatorial Code for Splicing Silencing: UAGG and GGGG Motifs
http://biology.plosjournals.org/perlserv/?request=get-document&#38;
doi=10.1371/journal.pbio.0030158

Posttranscriptional Gene Regulation by Spatial Rearrangement of the 3&#8242;
Untranslated Region
http://biology.plosjournals.org/perlserv/?request=get-document&#38;
doi=10.1371/journal.pbio.0060092
```

For the next step, we'll replace some HTML entities with their ordinary equivalents. This is a perfect use for a dictionary, a small portion of which is shown along with a function that uses it and a changed version of `print_atag` in Example 8-3. (The regular expression doesn't change.) With this code, the `′` in the last title in the previous output is replaced with a single quote.

Example 8-3. Extracting <a> links from an HTML file, step 3

```
html_entities = {
# Reserved characters in HTML
    '&#34;' : '"', '&quot;' : '"',
    '&#39;' : "'", '&apos;' : "'",
    '&#38;' : '&', '&amp;' : '&',
    '&#60;' : '<', '&lt;' : '<',
    '&#62;' : '>', '&gt;' : '>',
```

```
# ISO 8859-1 symbols      160-191, 215, 247
    ' ' : ' ', ' ' : ' ',
# Math symbols
    '&#8211;' : '-', '&ndash;' : '-',
    '&#8212;' : '-', '&mdash;' : '--',
    '&#8242;' : "'", '&prime;' : "'",
    '&#8243;' : "'", '&Prime;' : '"',
}

def replace_substrings(string, dictionary):
    """Return a copy of string in which every key of dictionary has
    been replaced with the corresponding value, in arbitrary order"""
    for key, value in dictionary.items():
        string = string.replace(key, value)
    return string
```

Although they didn't happen to show up in the one web page used for the preceding examples, experimentation with other web pages revealed that sometimes the value of an `href` attribute begins with a pound sign (`#`). These are document fragment markers used at the end of a URL to specify a location within the page containing the marker. We don't need these, so we want to exclude `<a>` tags whose `href` attributes have a value beginning with a pound sign.

We could exclude these tags by adding `\#` to the list of forbidden characters where the value of the `href` attribute is captured. (The backslash is required to prevent Python from interpreting the `#` as the beginning of a comment.) The problem is that although this would make the value match fail, it wouldn't make the overall match fail: all we'd get is an empty `href` value with whatever content the fragment identifier was associated with.

Nothing says we have to do all the work with a single regular expression, though. We can use one regular expression to extract something from the results obtained using another one, or we can filter the results we get back from a call to `findall`. To solve this problem we'll filter the results, as shown in Example 8-4.

Example 8-4. Extract <a> links from an HTML file, step 4

```
def get_all_atags(string):
    return [result for result in atagpat.findall(string)
            if result[0][0] != '#']
```

> The change is in the function `get_all_atags`, and it justifies having made this a separate function. When this was added during development of the earlier stages, it seemed entirely superfluous; there was no indication it would ever do anything other than a `findall`. It was placed there on principle, but now it turns out to have been a good decision. Note how this works—it's important.

The code should now give us what we want from the value of the `href` attribute in the `<a>` tag. Some of the values will be ugly because they form HTTP requests with arguments, as indicated by the presence of a `?` in the URL, and some of the request strings are quite long. For example, the URL corresponding to a link to "next" on the page with the PLoS search results looks like this:

```
?request=advanced-search&search_fulltext=1&row_start=11&
issn=15457885&surname_type=all&surname=&fname_type=all&fname=&aff_type=all&
aff=&fundsrc_type=all&fundsrc=&anywhere_type=phrase&
anywhere=nonsense+mediated+decay&title_type=all&title=&abstract_type=all&
abstract=&biblist_type=all&biblist=&tblfig_type=all&tblfig=&
jrn_issn=15457885&subj_id=all&vol_pubdate_start=&iss_pubdate_start=&
vol_pubdate_end=&iss_pubdate_end=&vol_no_start=&iss_no_start=&vol_no_end=&
iss_no_end=&limit=10&order=score&document_count=12#results
```

We could filter these, or perhaps give the program's user that choice, but since so many web pages offering document retrieval do so by way of requests rather than URLs linking to documents directly, we'll leave them in. That will allow the results to be used as real URLs, perhaps in an HTML page the code writes.

There's one other problem with this code that we'll put off dealing with until the next chapter: many URLs on a page will not begin with a site address (or a protocol). These are addresses relative to either the root of the site from which the page was retrieved or the directory from which the page was retrieved (the former is the case if the partial URL begins with a slash). For now, we won't be able to extract sufficient information from these URLs for them to be used to retrieve a page or submit a request.

Next, we'll turn our attention to the content extracted. The challenge here is that the content can contain all sorts of markup, and we want the output to be plain text. This is another case where it is too difficult to insert this additional processing into the existing regular expression. Instead, we'll use another one to remove the tags from the content and add that step to `get_all_atags`. We'll also add a test in `print_atags` to avoid printing empty content. The comprehension in `get_all_tags` would be a little messy if we didn't unpack the results of `findall`, so we add that in Example 8-5 too.

Example 8-5. Extracting <a> links from an HTML file, step 5

```
tagpat = re.compile(r"<[^>]+?>") # easy - anything within a pair of <>s

def remove_tags(content):
    return tagpat.sub('', content)

def get_all_atags(string):
    return [(url, remove_tags(content))
            for url, content in atagpat.findall(string)
            if url[0] != '#']

def print_atags(string):
    for atag in get_all_atags(string):
        if atag[1]:
            print_atag(atag)
```

More elaborate HTML pattern matching

Example 8-6 shows a simple program for printing an outline of a web page based on its heading tags (`h1`, `h2, ..., h6`). (Most modern web pages use other organizational techniques, so you might not find many—or any—heading tags on a page you download, but heading tags are still used for simple pages.) Try this function with the path to a simple web page with header tags that you've saved to the disk.

Example 8-6. Extracting an outline from an HTML file

```
pattern = r'<h(\d).*?>(.+?)</h'
def print_outline(filename):
    with open(filename) as fil:
        for level, hd in re.findall(pattern, fil.read()):
            # indent for each level and print the level in brackets
            print("{0}[{1}] {2}".format(' '*3*int(level), level, hd))
```

> When running Python from the command line, you may find that attempts to print Unicode characters cause errors. If so, this is because your terminal application doesn't support Unicode. In Unix environments, if you set the environment variable `LANG` to something like `en_US.UTF-8` before running Python you will be able to print Unicode characters. You won't run into this problem in IDLE, though, because its shell is built on top of code that supports Unicode.

Even if you can find a page that uses header tags for its general organization, there may be other markup around the header contents that ruins the output. For instance, go to *http://www.ncbi.nlm.nih.gov* and save the BLAST QuickStart document *bookshelf/br.fcgi?book=comgen&part=blast* to your disk. Then load the definition of `print_outline` and call it on the name of the file you used when you saved the document.

The output will be an outline, but it contains extraneous formatting detail—in particular, a lot of `span` tags. (`span` tags define formatting attributes for the content they surround.) Here's a sample line from that file:

```
<h4> <span class="mw-headline"> Documentation </span></h4>
```

We can filter out the `span` tags with a more complex regular expression, as shown in Example 8-7.

Example 8-7. Filtering span tags from HTML

```
pattern = re.compile(r'''
<h(\d)[^>]*?>   # a header start tag, such as <h2>
\s*             # optional whitespace
([^<]*)         # header text before the span
(<span[^>]*>)?  # optional span start tag -- a '<span', followed by
                # characters other than > until the first >
(.*?)           # header text after the span
(</span>)?      # optional span end (we aren't checking that this
                # ends the span tag found previously)
\s*             # optional whitespace
```

```
</h              # header end tag (we aren't checking that this ends
                 # the one just started)
                       ''',
                       re.IGNORECASE | re.DOTALL | re.VERBOSE)

def print_outline(filename):
    with open(filename) as fil:
        for level, pretag, optstart, posttag, optclose in \
            pattern.findall(fil.read()):
            print("{0}{1}{2}".format(' '*3*(int(level) - 1), pretag, posttag))
```

With some practice, you'll be able to do this sort of thing yourself. However, unless you have a lot of web pages to process that use tags in a similar way, your attempts to cope with all the details by writing increasingly complex code and regular expressions won't be worth the effort and frustration. Instead, either use several regular expressions or a mix of regular expressions and regular programming techniques.

Turning HTML into plain text

One important use for `str.replace` is to delete all occurrences of a substring from a string, by providing an empty string as the replacement. That is also an important use of `re.sub`. As a simple (though not entirely realistic) example, let's turn an HTML file into plain text by removing all the tags. The results will be pretty much a mess, since they'll reflect the spacing and line breaks of the original HTML rather than any kind of formatted text equivalent. More useful results could be obtained if the patterns were tuned to specific kinds of HTML pages.

Before we remove the tags we should skip to the `<body>` tag that starts the page's content, so we don't have to deal with the complexities of what can appear before that. There is also the complication that pages can contain script code placed between a `<script>` start tag and a `</script>` end tag. Although usually these occur in the `<head>`, they can occur anywhere in the `<body>` too. There can also be comment tags, which can get tangled up in the other tags. (Sound familiar? This is another "skip introduction.") Our code will proceed in these steps:

1. Skip to the `<body>` tag.
2. Remove all `<script> ... </script>` sections of the body.
3. Remove all comments (`<!-- ... -->`).
4. Remove all other tags.

We'll use `re.search` with the `re.IGNORECASE` flag to locate the `<body>`, `<script>`, and `</script>` strings, as just described. Normally we would define a separate function for each of these steps, but since each one after the first would do the same thing—one call to `sub`—we can do something a bit more clever: we can just loop through the patterns in succession in one function. This makes it easy to extend the program. For instance, if we also want to remove the contents of forms or all lines with just spaces, we can add another pattern for that to the patterns list. Example 8-8 demonstrates.

Example 8-8. Extracting plain text from HTML

```
patterns = (r'<!--.*?-->'),              # comments
            r'<script.*?</script>,       # scripts
            r'<[^<>]*>),                 # other tags
           )

def skip_to_body(string):
    matchobj = re.search('<body.*?>', string, re.I)
    return string[matchobj.end():] if matchobj else string

def remove_pats(string, pats):
    for pat in pats:
        string = re.sub(pat,' ', string, flags=re.I|re.S)
    return string

def remove_html(string)
    return remove_pats(string, patterns)
```

> HTML pattern matching is notoriously tricky. In HTML patterns, even nongreedy matches often consume too many of the target string's characters. If you look back at the regular expression patterns in the preceding examples, you'll see that there are a lot of places where a character set is used to *exclude* one or two characters. The excluded characters appear right after the repetition characters (for example, `[^>]*?>`, which matches everything up to the first `>`). Typically it is angle brackets that govern the structure of the match, but quotes and other delimiting characters play an important role too. If you are having trouble with a pattern that matches too much, try using this technique to restrict the repetitions it contains.

Structured HTML Processing

The techniques and examples we've seen so far in this chapter are fairly crude. Many kinds of complications arise that make programs using them limited in scope and brittle. In this section we consider the source of the problems and explore the use of a Python module that solves some of them.

Problems with HTML pattern matching

Extracting the contents of header tags is one of the simplest things you might do with an HTML document. Even so, the expressions we used to accomplish that were not simple. Attempts to use regular expressions to selectively extract other parts of HTML pages will often run into more substantive difficulties.

One of the most irritating problems is that the people or programs producing the web pages from which you are extracting data may change the markup slightly after you've written your code. Where there was a space, for instance, there might be a "nonbreaking space" entity, written in HTML as ` `. Some spaces may be eliminated, and line

breaks might be moved or deleted. Attributes in tags may also be added, removed, or changed. Every time this happens, you may have to update your program to accommodate the changes.

The real problem with HTML tags, though, is that they are nested. An `li` tag inside an `ol` tag can contain another `ol` tag that contains `li` tags, and so on. Likewise, there can be all sorts of `span` and `div` tags embedded inside other `span` and `div` tags. The documents are fundamentally tree-structured, like the example of the suffix tree in Chapter 4. Trees are recursive structures, so they can't be processed by linear methods.

In fact, browsers actually do convert the HTML they obtain from web servers into tree-structured data, based on a standardized "document object model." In that model, each element has attributes, content, and children, just like the tags we've been describing. Because elements have children, the document object model is organized as a tree.

There are basic computer science techniques for handling recursive structures other than the tree pattern we encountered in Chapter 4. However, these can't handle HTML generally, for the same reasons the other Python libraries can't: many start tags in many HTML documents do not have corresponding end tags, and many complex parts of HTML files don't fit the simple pattern of tags with an opening, optional content, and a closing. Browsers must deal with these complexities, but it's too big a challenge for ordinary programming.

Structured HTML parsing: html.parser

Fortunately, Python's library contains a class that *can* deal with messy HTML content. Contained in the `html.parser` module, `HTMLParser` defines a framework that calls empty methods at significant points in its parsing of HTML text. The framework is fully implemented, though by itself it does nothing. All you have to do is define a subclass of `HTMLParser` that overrides some of its methods with definitions that implement the actions you want your application to perform.

The way to use your subclass is to create an instance of it (with no arguments), then provide it text with the `feed(`*`string`*`)` method. This method implements a form of buffering, so you can actually call it multiple times. When you are done feeding the instance, you call `close` on it. Methods called by an `HTMLParser` instance when it encounters the corresponding HTML construct include:

`handle_starttag(`*`tag`*`, `*`attributes`*`)`
: Called when `<`*`tag`*`>` is encountered, with *`tag`* the name of the tag (in lowercase) and *`attributes`* a list of name/value pairs

`handle_endtag(`*`tag`*`)`
: Called when `</`*`tag`*`>` is encountered, with *`tag`* the name of the tag (not including the slash)

`handle_startendtag(`*`tag, attributes`*`)`
: Called when a tag that is closed in its start tag is encountered (such as `<hr/>`), with ***`tag`*** the name of the tag (in lowercase) and ***`attributes`*** a list of name/value pairs; if not overridden, it calls `handle_starttag` and then `handle_endtag`

`handle_data(`*`data`*`)`
: Called for text that is not a tag or something else special to HTML, with ***`data`*** a string containing that text

`handle_comment(`*`data`*`)`
: Called when a comment is encountered, with ***`data`*** the text of the comment

As a simple example, parsing `<h1>a heading</h1>` will result in the following calls:

```
handle_starttag('h1')
handle_data('a heading')
handle_endtag('h1')
```

There are a few other "handlers" that cover more obscure HTML items, which aren't mentioned here. Example 8-9 shows a subclass of `HTMLParser` that implements the outline generator in a more robust way than the one in Example 8-6.

Example 8-9. Using a subclass of html.parser.HTMLParser

```
class HTMLOutlineParser(html.parser.HTMLParser):
    """Show an indented outline of the headings in the HTML file provided as a command-line argument"""

    # HTMLParser converts tag to lowercase
    HeadingPat = re.compile('title|h([1-6])')
    Indent = 4

    def __init__(self):
        super().__init__()
        self.inheading = False

    def handle_starttag(self, tag, attrs):
        match = self.HeadingPat.match(tag)
        if match:
            self.inheading = True
            if match.group(1): # otherwise title & do nothing
                print('{0:{1}}[{2}]'.format(' ',
                                            self.Indent * int(match.group(1)),
                                            match.group(1)),
                      end=' ')

    def handle_data(self, data):
        if self.inheading:
            print(data, end=' ')
            # end='' so printing stays on this line in case
            # parts of the heading are nested inside other
            # markup, such as <i></i>

    def handle_endtag(self, tag):
        if self.inheading and self.HeadingPat.match(tag):
        # assuming heading tags (h1, h2, ...) cannot be nested
```

```
            print()             # close tag
            self.inheading = False

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print('Usage: prognam file.html')
    else:
        filename = testfile
        parser = HTMLOutlineParser()
        with open(filename, encoding='utf8') as file:
            contents = file.read()
        print('File contains', len(contents), 'bytes')
        parser.feed(contents)
        parser.close()
```

XML

XML stands for the *Extensible Markup Language*. It defines some basic syntactic rules for tags, attributes, and content that are similar to, but stricter than, those in HTML. What it doesn't define are what tags and attributes a document can use. That's the extensible part: XML documents can contain *any* tags, attributes, and content. The tag names, their allowable attributes, what kinds of tags can be nested in what other kinds of tags, and other such details are specified by a *schema* written in a formal schema language, such as Document Type Definition[‡] and RELAX NG.[§]

Before looking at Python's facilities for processing XML files, we'll explore their structure in more detail. We'll also take a look at an XML datafile that we'll use in subsequent Python examples.

The Nature of XML

Roughly speaking, XML is the data equivalent of HTML. Its purpose is to support the text-based representation of structured data in well-defined formats. A great deal of data in bioinformatics, as well as in other fields, is available in XML form on the Web.

XML must adhere to very strict rules:

- The first line of an XML file is expected to specify the schema it uses.
- Tags with textual content must be completed by corresponding end tags.
- Attribute values must be quoted (single or double).
- A tag with no content may end with `/>` in place of a separate end tag.

‡ See *http://www.w3.org/QA/Tips/Doctype* and *http://www.w3.org/QA/2002/04/valid-dtd-list.html*.

§ See *http://www.relaxng.org*.

The regularity enforced by these rules makes parsing XML files straightforward. That doesn't mean you'll want to write that code yourself, though, especially not when there are Python library modules to do it for you.

Other representation languages used in bioinformatics, such as the *Resource Description Framework* (RDF)[‖] and the related *Web Ontology Language* (OWL),[#] are based on XML. Those languages are part of the more general Semantic Web[*] initiative. Various parsers and even logical deduction mechanisms are available for them.

A variety of approaches to parsing XML files are possible. Some parsers process an entire file at once, while others parse incrementally. Some produce a tree structure that represents the XML content, while others generate "events."[†] (The subclass of `html.parser.HTMLParser` defined in Example 8-9 implements simple event-based processing.) The event-based model is particularly useful when a large amount of XML text is to be searched for specific information. The tree-based model is appropriate where many actions will be performed on the results of parsing the entire file. Some modules can be used in several ways, and most have advanced features we won't cover here.

A parser understands a language's grammar, but it leaves it up to other code to process the syntactic elements it has extracted from its input. Python's library contains several event-based parsers, and XML's highly constrained syntax makes parsing an XML file straightforward. (Verifying that a schema's rules are followed in addition to the basic syntactic rules is a much more difficult challenge.) Parsing an XML file basically generates information about tag names, attribute name/value pairs, tag content, and the nesting relationships among the tags. It is up to the application to interpret what all that means.

Our main goal here is to demonstrate the nature of XML data and some of the ways it can be processed. We'll use two Python modules to parse some XML data downloaded from the Web. One, `xml.etree.cElementTree`, creates a tree for subsequent use. The other, `xml.parsers.expat`, is an interface to the event-based Expat C library.[‡] The

‖ See *http://www.w3.org/RDF*.

\# See *http://www.w3.org/TR/owl-features*, *http://www.w3.org/TR/owl-guide*, *http://www.w3.org/TR/owl-ref*, and (for the new features introduced by OWL 2) *http://www.w3.org/TR/2009/WD-owl2-new-features-20090421*.

* See *http://www.w3.org/2001/sw/*.

† We've seen this kind of choice before, in processing FASTA files. We had the option of reading in all the sequences with their descriptions and then searching for sequences meeting certain criteria, or reading one sequence at a time and stopping after locating the one(s) meeting those criteria. The advantages and disadvantages here are the same. Processing an entire file means doing a lot of computation and producing a large data structure that the program can then use for repeated actions. An event-based scan of a file could potentially stop after a relatively small amount of processing and retrieve only a small amount of data; however, getting different data later would mean reparsing the file.

‡ See *http://www.libexpat.org*.

`cElementTree` module will detect violations of the XML file's schema, but Expat will not—it is a *nonvalidating* parser. The `xml` package contains several other parsers, including some that work with the same *document object model* (DOM) that browsers use when interpreting HTML and executing JavaScript, but we won't cover those here.

An XML File for a Complete Genome

For this part's examples we are going to use an XML file representing an entire bacterial genome. The steps followed to obtain this are as follows (you can choose your own organism, of course; we're using a microbe genome because its XML file is of manageable size):

1. In a browser, go to *http://www.ncbi.nlm.nih.gov/genomes/MICROBES/microbial_taxtree.html*.
2. Expand some taxa (in green) until you see individual organism names (in blue), e.g., the class *Acidobacteria*, about nine lines down.
3. Click on an organism name. For *Acidobacteria* only one organism name is shown: *Acidobacteria capsulatum*.
4. Click on the organism to get a page with an extensive summary of the information available about the organism and its genome.
5. Below the tabs visible in the upper-left corner, select "Protein Table" from the pull-down that is showing "Overview."
6. From the "Send to" pull-down, select "File."
7. Save the file wherever you want, perhaps naming it with the "UID" of the organism, as shown in the page title, and the extension *.xml*.

This will be a fairly large file. (The one for *Acidobacteria capsulatum* was 44 MB.) It has a rather deeply nested structure, which is summarized next. While you read the summary, it will help to refer to the display in the web page from which you downloaded the data to see how the two correspond.

The outline of the saved file is as follows. The numbers of lines listed are rough counts for the file *Acidobacteria capsulatum*, which identifies 3,377 proteins. The file consists of:

- The two lines identifying the XML version and the document schema that usually appear at the beginning of an XML file
- A few top-level start tags
- Some tags inside a `Bioseq-set_descr` tag that provide basic information about the organism, its source, publication data, etc. (1,500 lines)
- Some tags inside a `Bioseq-set_seq-set` tag that provide information about the genome (142,000 lines)

- A `Seq-entry` tag for each protein that describes that protein's sequence (600,000 lines)
- A `Seq-annot` tag for each protein that describes its annotations (155,000 lines)

The file uses a over 200 distinct tags and over 500,000 tags in total, as well as some attributes and tag content.

> The following Python script counted the tags and distinct tags. It is very important to appreciate the power of the tools you are acquiring and learn to use them not just for what you consider "programming," but for small one-time tasks as well):

```
import re
pat = re.compile((r'</([A-Za-z-_]+)')
with open(filename) as file:
    closetags = pat.findall(file.read())
print('The number of total tags was {}, ' +
      'of which {} were distinct' .format(len(closetags), len(set(closetags)))
```

The ElementTree Module

The `xml.etree.ElementTree` module provides an easy way to read an entire XML document and create objects to represent its elements, attributes, and content. Its only disadvantage is the same as its strength: it reads the entire file. Whether to use an event-based parser or one that processes the entire file and creates a data structure from its contents depends on what your application will do with the parsed XML. If you had to process an extremely large XML document, or use the objects for many smaller ones together, reading their entire contents could cause problems. Searching for a particular element, for example, would suggest the use of an event-based parser.

For extreme situations or complicated processing patterns, one of Python's XML libraries that we aren't covering here might be more appropriate. There are also a few more advanced features of the `ElementTree` module that would allow partial processing to get something of a compromise between the two approaches.

Two Versions of the ElementTree Module

There are actually two versions of the `ElementTree` module: `xml.etree.ElementTree` and `xml.etree.cElementTree`. The first is written in Python, and the second is written in C with a Python interface. The C version's parser is quite a bit faster than the Python one, and the behavior of some of its methods differs in a few advantageous ways. In particular, its functions that return requested nodes return generators that produce them one at a time, whereas the Python versions return complete lists. For large numbers of nodes, a generator would be much more efficient. There being in this case no particular advantage to using the Python version or disadvantage to using the C version, we'll use the C one.

To avoid confusion between the *module* `ElementTree` and the *class* `ElementTree.ElementTree`, both in this discussion and the code examples, we'll import the module with the name `Etree`. In addition to being a shorter name, creating this alias for the actual module has the more significant benefit that the code will be able to work with either version of the module—we just have to change the `import` statement to import from whichever of `xml.etree.ElementTree` and `xml.etree.cElementTree` we want to use:

```
>>> import xml.etree.cElementTree as ETree
>>> ETree
<module 'xml.etree.cElementTree' from
'/usr/local/lib/python3.1/xml/etree/cElementTree.py'>
```

Getting started with ElementTree

An instance of the class `ETree.ElementTree` represents a tree with features corresponding to the various aspects of XML. The tree consists of `ETree.Element` instances. Most of the `ETree.ElementTree` instance methods simply call the same method on its root `Element`—an example of delegation, as discussed in "Class decomposition" on page 186 (Chapter 5)—but a few provide additional functionality.

The module function `ETree.parse(`*`source`*`)` parses the XML text from *`source`*, a filename or a file open for reading, and returns an instance of `ETree.ElementTree`. To write a tree to a file, call its method `write(`*`destination`*`)`, where *`destination`* is either a filename or a file open for writing.

`ElementTree`s are quite general and can be used for pretty much any kind of tree-structured information. The module provides functions for building a tree programmatically, which you can use to convert hierarchical but non-XML data into XML, or to structure data your program is generating in preparation for its output as XML. You could also use it to construct trees for the program's own internal purposes that have nothing to do with XML. These are powerful capabilities, but we won't be discussing them any further. Our focus here is on manipulation of the tree created by the parsing functions.

We'll start by creating a tree for an XML genome file, such as one downloaded as described earlier in "An XML File for a Complete Genome" on page 302. We can see in Example 8-10 that `ETree.parse` returns an instance of `ElementTree`. The root of the tree is an `Element`. We can also see that while `ElementTree` instances don't give us anything more than the default information when the interpreter prints them, `Element` instances at least show the names of their tags.

Example 8-10. Basic ElementTree manipulations

```
>>> source = 'data/Entrez/sequences-24289.xml'
>>> tree = ETree.parse(source)
>>> tree
<ElementTree object at 0x18a3e70>
>>> root = tree.getroot()
```

```
>>> root
<Element 'Bioseq-set' at 0x18a7188>
```

The XML-related features of an `Element` are accessed as Python attributes, not with method calls. There are four:

`tag`
: The name of the tag, as shown in the printout in Example 8-10

`attrib`
: A dictionary of attribute names and values as extracted from the start tag

`text`
: The text, if any, that appeared between the tag and its end tag

`tail`
: The text appearing between the end tag and the start of the next tag, whatever that tag is

The Python documentation warns that although the current implementations use actual Python dictionaries for `Element.attrib`, that may not always be the case. It is recommended that the following `Element` methods be used instead of directly accessing `attrib`. They are defined to perform the same actions as the corresponding dictionary methods:

- `items()`
- `keys()`
- `values()`
- `get(`*`key`*`[, `*`default`*`])`
- `set(`*`key`*`, `*`value`*`)`

Navigating around an ElementTree

Pretty much everything a program does with a tree involves navigating around its nodes. Sometimes a program takes steps to get to a specific node. Sometimes it searches for one that meets specific criteria. Other times it *traverses* the entire tree, doing something to each node encountered (a Recursive Tree iteration, as described in Chapter 4). `ElementTree` and `Element` provide the following methods for these kinds of operations:

`find(`*`target`*`)`
: Returns the first (immediate) child matching ***`target`***

`findall(`*`target`*`)`
: Returns a list of all the (immediate) children matching ***`target`***, in the order in which they appear in the document

`findtext(`*`target`*`[, `*`default`*`=None])`
: Returns the text of the first (immediate) child matching ***`target`***, or ***`default`*** if no such element was found; an empty string is returned if a matching target is found that has no text content

`getiterator([`*`tag`*`=None])`
: Returns a generator that produces, from an element and all its descendants, all elements whose tag is *`tag`*; if *`tag`*`=None` or *`tag`*`='*'` the generator returns all of the elements

In addition, `Element`, but not `ElementTree`, provides the following method for navigating the tree recursively:

`getchildren()`
: Returns a list of all the subelements of `Element`, in the order in which their tags appeared in the document

`ElementTree`, but not `Element`, also provides the `getroot()` method to get its root `Element`.

Example 8-11 shows a function based on the Filtered Count template from Chapter 4 ("Nested iterations" on page 126). The definition counts the number of nodes in an `ElementTree` subtree whose tag has a specified name.

Example 8-11. Counting the nodes for a specified tag

```
def count_nodes(element, tagname):
    """Return the number of tagname nodes in the tree rooted at element"""
    count = 0
    for node in element.getiterator(tagnames):
        count += 1
    return count
```

Initial examination of the XML seemed to indicate that there was a series of `Seq-entry` tags, first one for the entire genome and then one for each protein. These contained core information. Later in the file there is a series of `Seq-feat` tags that follow in the file with sequence annotations. Let's see whether there are the same number of `Seq-entry` tags as `Seq-feat` tags:

```
>>> count_nodes(tree, 'Seq-entry')
3379
>>> count_nodes(tree, 'Seq-feat')
10360
>>>
```

Apparently not. This is an easy way to interactively investigate simple hypotheses you might have about the structure of a deeply nested and/or large XML file after some preliminary examination using an editor. How about getting all tag names in the file, avoiding duplicates by using a set?

```
>>> tags = set()
>>> for element in tree.getiterator():
```

```
        tags.add(element.tag)
>>> len(tags)
208
```

Let's count how many times each tag occurs:

```
>>> counts = {}
>>> for element in tree.getiterator():
        counts[element.tag] = 1 + counts.get(element.tag, 0)
```

Since there are 208 keys in this dictionary, we don't want to print the whole thing. We can first check the results obtained this way against the results obtained earlier using `count_nodes`:

```
>>> counts['Seq-entry']
3379
>>> counts['Seq-feat']
10360
```

Now let's print out the counts of the 20 most common tags. This is another way to explore the nature of an XML file. There's no guarantee that a frequently occurring tag is only used inside a particular other tag—it could be used all over the file—but at least these counts are a hint. With some more work we could write a program that counted how many times each tag appeared within each other tag, and so on:

```
>>> sortedtags = sorted(counts.items(),
                        key=lambda item: item[1],
                        reverse=True)
>>> for item in sortedtags[:20]:
        print(item)
('Object-id', 27150)
('Seq-id', 23872)
('Object-id_str', 20277)
('Seq-id_gi', 17116)
('User-field', 16898)
('User-field_data', 16898)
('User-field_label', 16898)
('Seq-loc', 13739)
('Seqdesc', 13520)
('SeqFeatData', 10384)
('Seq-interval_to', 10361)
('Seq-interval_id', 10361)
('Seq-interval_from', 10361)
('Seq-loc_int', 10361)
('Seq-interval', 10361)
('Seq-feat_location', 10360)
('Seq-feat_data', 10360)
('Seq-feat', 10360)
('Na-strand', 6984)
('Seq-interval_strand', 6984)
```

Here are a few examples of method calls using the tree created in Example 8-10:

```
# all elements with protein names
>>> proteins = root.getiterator('Seqdesc_title')
>>> protein1 = next(proteins)
```

```
>>> protein1                # first protein is actually the whole genome
<Element 'Seqdesc_title' at 0x1965218>
>>> protein1.text
'Acidobacterium capsulatum ATCC 51196, complete genome'
>>> next(proteins).text
'YkgG family protein [Acidobacterium capsulatum ATCC 51196]'
>>> next(proteins).text
'iron-sulfur cluster binding protein [Acidobacterium capsulatum ATCC 51196]'
>>> prot = next(proteins)
>>> print_element(prot)
 Seqdesc_title: cysteine-rich domain protein [Acidobacterium capsulatum ATCC 51196]
```

Example 8-12 shows a pair of function definitions used to print a subtree starting at a given `Element`. This is a straightforward use of the Recursive Tree Iteration template like the one in Example 4-26 (see "Recursive iterations" on page 128).

Example 8-12. Functions for printing subtree of an ElementTree

```
def print_element(element, level=1):
    """Print tag, content, and attributes of element, indented level
    spaces, but only if  element has text or attributes"""
    # ignoring tail
    if element.text and not element.text.isspace():
        print(' '*level, element.tag, ': ', element.text.strip(), sep='')
    elif element.attrib:
        print(' '*level, element.tag)
    for attr in sorted(element.keys()):
        print(' '*(level+2), attr, '=', element.get(attr))

def print_subtree(element, level = 0):
    """Print the tags, attributes, and contents of element and all of its children,
    in depth-first order, starting with an indentation of level spaces"""
    print_element(element, level)
    for subelt in element.getchildren():
        print_subtree(subelt, level+1)
```

Example 8-13 shows what a printout using these functions would look like.

Example 8-13. Printing an ElementTree subtree

```
>>> descrs = root.getiterator('Seqdesc_source')
>>> print_subtree(next(descrs))
Seqdesc_source
 BioSource
  BioSource_genome: 1
     value = genomic
  BioSource_org
   Org-ref
    Org-ref_taxname: Acidobacterium capsulatum ATCC 51196
    Org-ref_db
     Dbtag
      Dbtag_db: taxon
      Dbtag_tag
       Object-id
        Object-id_id: 240015
```

```
    Org-ref_orgname
     OrgName
      OrgName_name
       OrgName_name_binomial
        BinomialOrgName
         BinomialOrgName_genus: Acidobacterium
         BinomialOrgName_species: capsulatum
      OrgName_mod
       OrgMod
        OrgMod_subtype: 2
           value = strain
        OrgMod_subname: ATCC 51196
       OrgMod
        OrgMod_subtype: 35
           value = culture-collection
        OrgMod_subname: ATCC:51196
       OrgMod
        OrgMod_subtype: 255
           value = other
        OrgMod_subname: type strain of Acidobacterium capsulatum
      OrgName_lineage: Bacteria; Acidobacteria; Acidobacteriales;
Acidobacteriaceae; Acidobacterium
      OrgName_gcode: 11
      OrgName_div: BCT
```

Many other manipulations are possible using the small set of methods documented earlier. Experiment with them using a tree like this. While experimenting, it is best to assign names to results rather than just having them print in the interpreter—you never know how many thousands of lines will result from a method call! If you assign a name, you can check its length, or its first element, or some other feature before trying to print anything. You could also write functions that print only a limited number of items.

Example 8-14 illustrates a function that prints the information about a limited number of proteins starting at a specified number. This calls the `print_subtree` function of Example 8-12 and relies on the information that each protein's description's outermost tag is called `Seq-entry`.

Example 8-14. Printing information for a range of proteins

```
def describe_proteins(tree, limit=2, start=1):
    # start at 1 because 0 is the whole genome!
    iter = tree.getiterator('Seq-entry')
    # +1 to always skip entry for entire genome
    for n in range(start+1):
        next(iter)
    for k in range(limit+1):
        print('{:4}'.format(k+start))
        print_subtree(next(iter), 6)
```

Event-Based Processing

Before exploring one of Python's event-based XML parsers, we need to take an excursion to confront some unusual properties of event-based processing in general.

Function calls, exceptions, and the call stack

The usual pattern of function calling has the following characteristics:

1. When function `A` calls function `B`, it passes values for function `B`'s parameters.
2. Function `B` returns a value to function `A`. (Either it executes all of its statements without encountering a `return` statement, in which case it returns `None`, or at some point it executes a `return` statement.)
3. When function `B` returns, function `A` continues from where it called function `B`.

Function calls are implemented by adding a data structure called a *frame* to the *call stack* for each function called. That frame contains (at least) the following critical information:

- Bindings for the function's local variables
- The address in the computer's memory of the machine-level instruction at which the function should continue when a function it has called returns to it
- The frame it should return to when it executes a `return` statement or finishes executing all of its statements

These stack frames implement the usual pattern of function call and return. When a function returns, its frame is "removed" from the stack. Actually, the only thing that really happens is that an index representing the current top of the stack is incremented so that it references the previous stack frame. If the function of the previous frame then calls another function, the new function's frame simply overwrites the frame of the one that just returned. Figure 8-1 illustrates the process.

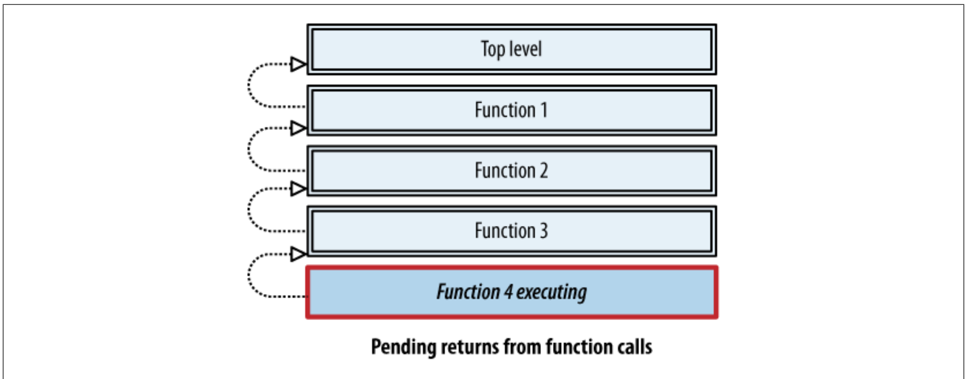


Figure 8-1. Function calling and stack frames

When an exception is raised, it returns control to somewhere further back in the stack. Unless the exception is caught within the same function in which it was raised, frames are removed from the stack, in a process called *unwinding*, until an `except` clause is encountered that matches the exception raised. This breaks the normal pattern of function call and return, since some of the "deeper" functions are simply abandoned without having a chance to execute the rest of their statements and return normally.

In the normal operation of the stack, each function returns control to the one that called it by making the caller's stack frame the current one. For control to return to the `except` clause of a `try` statement requires that there be another kind of entry in the stack to represent the clause apart from the stack frame of the function that contains the statement. When an exception is raised, Python unwinds the stack, checking every `except` clause entry along the way until it finds one that specified an exception class that matches the type of the exception being processed. Finally, it starts executing the statements in that `except` clause.

Almost everything about exception handling is the reverse of function calling:

- Each function call *adds one frame* to the stack, whereas raising an exception *removes one or more frames from the stack*.
- Function calls pass control to a *known* piece of code, whereas raising an exception tosses control up the stack with *no idea* of where it will end up.
- In the absence of an exception that bypasses the function call, a function *returns control to where it was called from*, whereas control *is never returned to where the exception was raised*.

In effect, raising an exception "calls" backward in the stack with a single argument to an unknown location, causing all the intervening stack frames to be discarded (see Figure 8-2). This is a strange beast indeed, yet one that plays critical roles. The lack of a direct connection between where the exception is raised and where it is caught, and the reverse direction of the frame movement in exception handling, is what makes it such a valuable tool.

The primary benefit of an exception handling mechanism is that it allows the separation of the code that detects an exceptional situation from the code that knows what to do about it. In particular, it allows library code to raise exceptions that are caught by application code that has called into the library. The library code that detects the exceptional situations will rarely know what actions it would be appropriate to take, while the code using the library will rarely be able to detect the exceptional situations.

Callbacks: Reversed function calls

There is another kind of reversed function call, known as a *callback*. It is not disruptive the way exceptions are: it is really just an ordinary function call used in a special way. Callbacks are a major feature of event-based processing where the processor, typically something from a library, calls a specific application function each time a corresponding

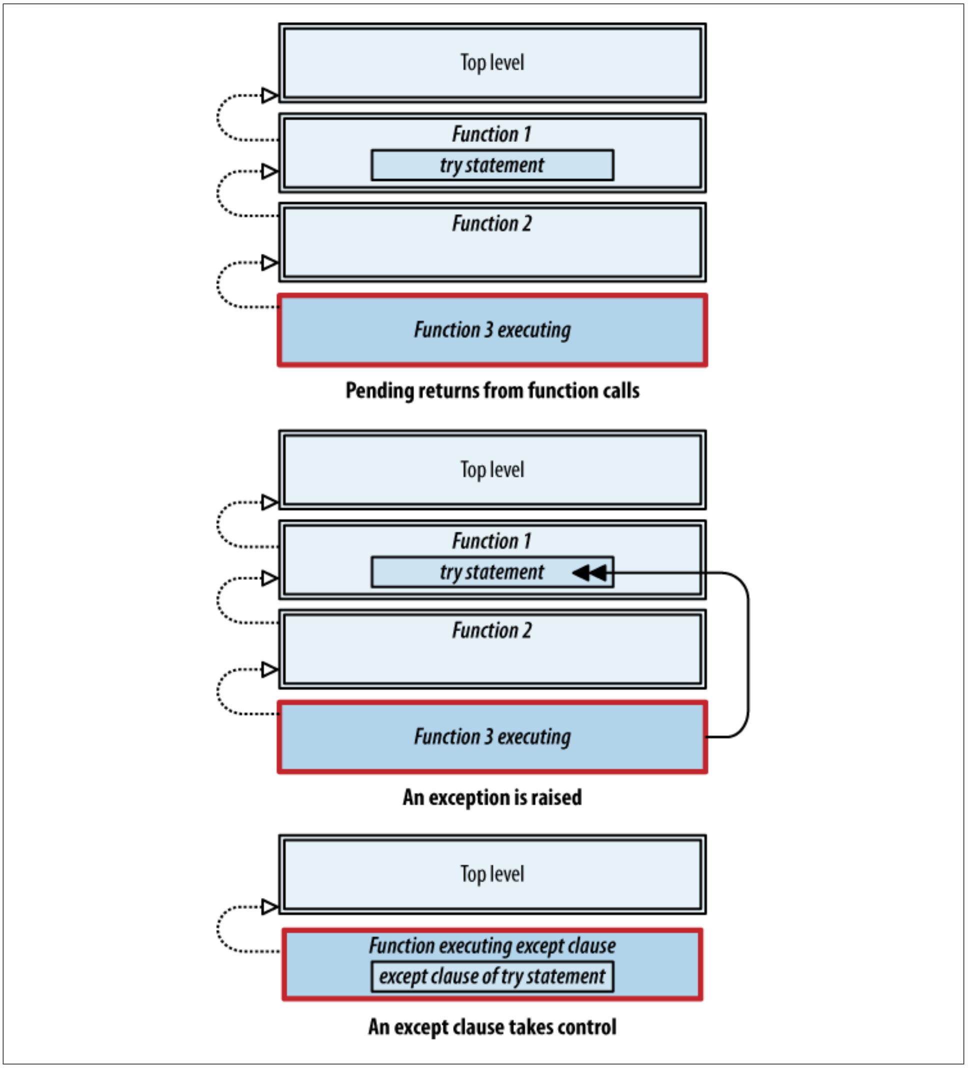


Figure 8-2. Stack frames and exceptions

event occurs during its processing. This places the processor in control of the computation while still allowing the application to define the actions to take at well-defined points.

Graphical user interface systems are the archetypal example. These systems monitor input devices such as mice and keyboards, constructing objects representing "events" for each keystroke and mouse click. It looks up each event in a table (a dictionary or something similar) to retrieve a description of the action to take in response to the event, usually expressed as a function to call. One of the challenges of this kind of architecture

is giving application code a way to specify which function should be called when a particular icon is clicked, key is pressed, and so on.

Callbacks are the mechanism that allows the application to respond to processor events. The application defines functions and binds events to them. Then it turns control over to the event-based system. When an event occurs, the system "calls back" to the application via the function corresponding to the event. The technical definition of the system and its callback mechanism specifies what arguments will be passed to the function call through this mechanism. Some systems also allow applications to register additional values to be passed back to a function when the corresponding event occurs.

No new mechanisms are required to implement a callback system. Event-based processing simply reverses the roles of library and application code. The library provides functions for registering callback functions. After the application registers its callbacks with the event-based system, it passes control to it. Subsequently, the application functions play the kind of passive role that library functions usually do. The order in which they are called is undetermined since the calls are in response to input to the event-based system. Figure 8-3 illustrates this arrangement.

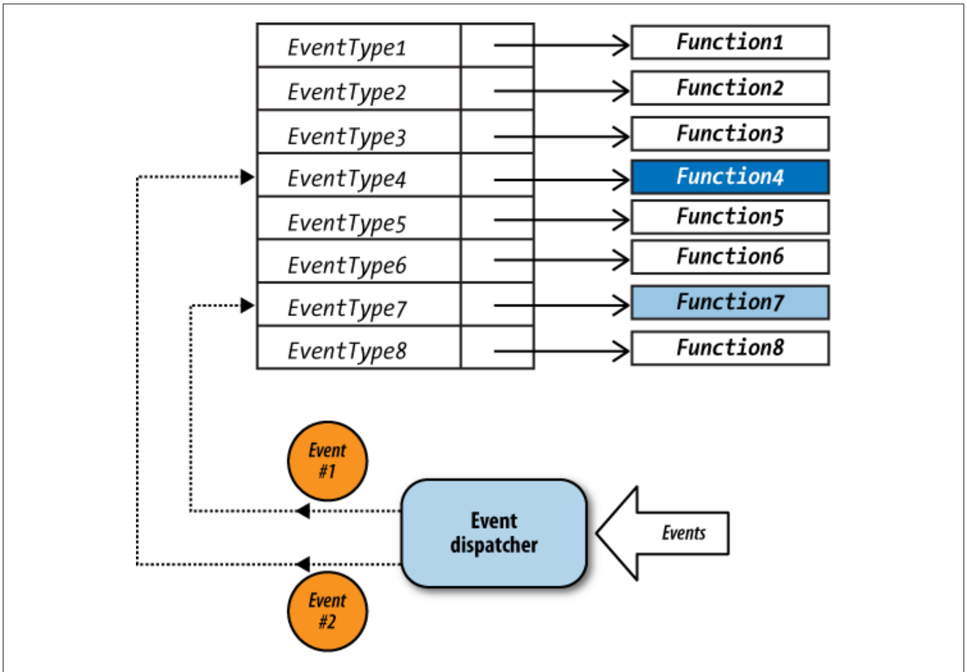


Figure 8-3. Event-driven architecture

> Once you understand all of this, you'll see that the major difference in programming an application with an event-based system is the initial setup that associates functions with events. Some systems use assignment statements to do the binding, while others use function calls. Either way, the principle is the same.
>
> Note that callbacks don't have to be functions named with a `def` statement—they can be "anonymous" functions created with a lambda expression. Just remember that lambda expressions cannot themselves include statements, though there's no problem with them calling functions that do. Like nested function definitions, lambda expressions have access to all the names in the scope in which they are defined. Therefore, they can pass information from the scope they are in as an argument to the function in another scope.

In a way, the `html.parser.HTMLParser` class we looked at earlier in this chapter implements event-based processing. Every time one of its instances encounters a tag in the HTML text it is parsing, it calls the method `handle_starttag`. Every time it encounters a closing tag, it calls `handle_endtag`. To implement an application using this class, you implement a subclass and override the class's "do-nothing" methods with definitions that perform your application's actions. This is a kind of event-based processing, with each kind of thing the parser recognizes causing the corresponding method to be invoked.

It feels as if `html.parser.HTMLParser` is in control, not your class, because it does all the parsing and event handling; your subclass just defines what to do with each event. In reality, though, the instance of your subclass does the parsing and event handling. It uses methods inherited from `HTMLParser`, including ones you may not even know about, but there is only one object doing the work. (In true event-based processing there is one object that generates events and calls event handlers and a separate object that handles the events.)

Programming for an event-based processor

Event-based architectures confront the programmer using them with two interesting challenges:

- How to stop the processing
- How to maintain application state between callbacks

These two problems stem from the same source. Typical applications are coherent in the sense that their conditionals, loops, iterations, and function calls constitute a seamless flow of computation. However, the application side of event-driven systems is fragmented: its functions are called by other code, at unpredictable times, and under unpredictable circumstances.

To set things up so an application exits in response to a certain event, you can simply have it call `sys.exit` from the callback function. However, it is sometimes the case that the application wants to take back control from the event-based system. Doing so requires returning past where the event loop was invoked, back into the application code. We can do this by raising an exception in a callback and catching it where the event system was invoked:

```
def stop(args):
    raise StopIteration

# set up callbacks, etc.
 try:
    call_external_event_loop()
 except StopIteration:
    pass
# do more
```

The code is straightforward, except that its `try` statement looks misleadingly like a typical one designed to handle errors. `StopIteration` was chosen as the exception class instead of the more general `Exception` to provide at least a hint of the purpose of the `try`. The strange function `stop`, which simply throws the exception, would be registered as the callback for an event. Unless something in the event processor happens to catch `StopIteration`, as soon as `stop` is invoked from the event loop it will throw the exception. The stack will then be unwound all the way through the event loop and back to the `try` statement that invoked it—awkward, but effective.

The second problem requires a way for separate functions to share state. Normally, names are bound to values through function calls and assignment statements. The names in one function are independent of the names in any other function, even if they are spelled the same. Likewise, names assigned inside a function are independent of names assigned at the top level of a module that contains it. Event-based architectures present a problem for this kind of arrangement because the application functions don't call each other, but instead are called by the external system. They are therefore denied the function call arguments mechanism by which they normally pass information to each other.

We addressed this problem earlier by using a class to encapsulate all the necessary state together with the methods that manipulate it. Many methods of the GenBank parser class we created in Chapter 5 (see Example 5-4 and the surrounding discussion) used the same variables. Instead of cluttering up the code with many parameters in method definitions and arguments in method calls, all the definitions just accessed the fields of the instance. A method of a subclass of `html.parser.HTMLParser` may need values computed in a previous call or in a call to a different event-handling method. Storing values in the fields of an instance makes them available to all the methods of the instance's class, whatever happens between calls to them.

`GenBankParser` was an ordinary class that defined all its own methods. The `HTMLOutlineParser` class that we defined in Example 8-9 is a subclass of the class that

generates the events. In both cases, all calls stay within the context of a single instance. When an external event-based processor is used, we need a way to tie the event handlers to instance state. Python gives us two easy ways to do just that:

- *Define one function inside another* (usually with a lambda expression, but one `def` statement can appear inside another). The inner function, when executed, has full access to the local state of the function in which it was defined. If the outer function is a method, the inner function can access the instance through the name `self`, just as the outer method can.
- *Use a bound method*. You have undoubtedly seen interpreter output of the form:

  ```
  <bound method ClassName.method_name of object>
  ```

  This is what you get when you type

  ```
  >>> name.method_name
  ```

  with no parentheses, where ***`name`*** is the name of an instance of ***`ClassName`*** and ***`method_name`*** is the name of a method of ***`ClassName`***. The way this works might surprise you at first, but it is just binding at work as usual. Because ***`method_name`*** is accessed through the value of an instance, its value is the value that ***`name`*** has within the instance's namespace, with **`self`** bound to that instance. In other words, a bound method carries around with it the instance to which it is bound.

Example 8-15 presents a tiny class that demonstrates how these two approaches work.

Example 8-15. Access to outer scope from inner functions

```
class Value:
    """Hold a value, demonstrating some aspects of nested scope"""
    def __init__(self, v=None):
        self.val = v

    def get(self):
        return self.val

    def set(self, v):
        self.val = v

    def setter(self):
        """Return a function that can be called to change the value held by the instance"""
        return lambda newval: self.set(newval)

    def __str__(self):
        return str(self.get())

    def __repr__(self):
        return 'value({})'.format(self.get())

>>> v = Value()
>>> v
value(None)
>>> v.set(0)
```

```
>>> v
value(0)
>>> fn = v.setter()                    # with parentheses
>>> fn(9)
>>> v
value(9)
```

The function returned by `Value.setter` is an ordinary function with one parameter, not a method. When it is called, it calls `self.set` with its parameter as the argument. The point of this is that every function (and method) has access to all the names available where it is defined. If it is defined inside another function (or method), it has access to that other function's parameters and to any names the other function has assigned. This demonstrates the first of the previously mentioned techniques. Using the same `Value` class, here's a demonstration of the second technique:

```
>>> fn = v.set                         # no parentheses
>>> fn
<bound method value.set of value(9)>
>>> fn(4)
>>> v
value(4)
```

expat

The `xml.parsers.expat` module is a straightforward incremental event-based parser based on the widely used Expat C library.§ To use it, you define a few callback functions, create an instance of a parser object, assign some of its fields to your callback functions, and feed the parser text. The text can be fed all at once or in segments. To create a new instance, use the following method:

`xml.parsers.expat.ParserCreate([`*`encoding`*`])`
: Creates an instance of `xml.parsers.expat.xmlparser` for text with the encoding specified by ***`encoding`***. ***`encoding`*** defaults to `None`, with the only other options being `'UTF-8'`, `'UTF-16'`, `'ISO-8859-1'` (Latin1), and `'ASCII'`.

To parse XML text, use one of the following two methods with the instance returned from `ParserCreate`:

`Parse(`*`string`*`[, `*`endflag`*`])`
: Parses ***`string`***, which may be empty; ***`endflag`*** (default `False`) must be `True` on the last call to this method

`ParseFile(`*`readable`*`)`
: Parses the contents of ***`readable`***, which can be any object that provides the method `read(`*`nbytes`*`)`; since it reads bytes, if ***`readable`*** is an opened file, it must have been opened in binary mode

§ See *http://www.libexpat.org*.

The most common event handlers are similar to the event-handling methods of html.parser.HTMLParser, because HTML is similar to (though much less well-structured than) XML. However, they are *fields*, not *methods*: you *assign* them rather than overriding them in a subclass. Their values are functions.

> Remember, functions (including lambda expressions) are objects, objects are values, values can be named, and fields of an instance are one kind of name. A function can therefore be assigned as the value of a field of an instance.

The three event handler fields that will serve most needs are as follows, with parser an instance of xml.parsers.expat.xmlparser:

StartElementHandler
: The function to be called when a start tag is encountered, with its first argument the name of the tag and its second a dictionary of attribute names and values

StopElementHandler
: The function to be called when an end tag is encountered, with the name of the tag as its sole argument

CharacterDataHandler
: The function to be called for text encountered outside of a start or end tag, with the text as its sole argument

Other than callbacks, the instance has a number of fields that control its behavior, most of which are fairly technical. This one is worth knowing about:

buffer_text
: Controls whether text parsing should stop at every newline (if False, the default), calling CharacterDataHander, or should join consecutive text lines together and pass them as one large string to CharacterDataHander; this can be set at any time, even after some text has been parsed, and can make parsing considerably more efficient

Obtaining a single piece of information

We'll start with something simple. We downloaded this enormous file, but we want to extract the content of just one tag from it. Example 8-16 shows how we would extract the value of the first (and, as it turns out, the only) occurrence of the tag <Seqdesc_title>. It uses the exception handling trick discussed earlier to stop the program. The program is simple because it doesn't need to keep state between callbacks, and all it does when it receives the StopIteration exception is print the information that was included in the exception when it was created. This assigns event handlers to lambda expressions inside its function definitions: while the parameters of a particular kind of handler are predefined, the lambda expression doesn't have to use them all, and it can call other functions with any combination of argument values.

Example 8-16. Getting content from a specified XML tag

```
import xml.parsers.expat

def handle_start(p, target, name):
    """Install data handler when target tag is encountered"""
    if name == target:
        # now install handler for data tag
        p.CharacterDataHandler = lambda data: return_data(data)

def return_data(data):
    """Stop searching, since the target has been found;
     installed when the target start tag has been encountered"""
    raise StopIteration(data)

def lookup(filename, target):
    p = xml.parsers.expat.ParserCreate()
    # install handler for start tags
    p.StartElementHandler = \
        lambda name, attrs: handle_start(p, target, name)
        # one way to "forward" a parameter to an outside function

    with open(filename) as file:
        try:
            while True:
                p.Parse(file.read(2000))      # read 2000 bytes at a time
        except StopIteration as stop:         # expected!
            return str(stop)
```

Extracting a few pieces of data

Next, we'll extract the genus and species names of the organism whose genome is in the file. The XML for this looks like:

```
<OrgName_name_binomial>
  <BinomialOrgName>
    <BinomialOrgName_genus>Acidobacterium</BinomialOrgName_genus>
    <BinomialOrgName_species>capsulatum</BinomialOrgName_species>
  </BinomialOrgName>
</OrgName_name_binomial>
```

Example 8-17 shows an implementation similar to that in the previous example. What's different about this one is that we can't just stop when we find one tag—we need to find a second one too (first the genus, then the species). Because we need to maintain state between callbacks, we must define a class and assign the callback fields to methods of one of its instances. The code starts by installing the typical three `expat` parser callbacks. This demonstrates the way `expat` is typically used.

Example 8-17. Getting the content of several XML tags

```
import xml.parsers.expat

class GenomeBinomialParser:

    ChunkSize = 4000                # number of bytes to read at a time

    def __init__(self):
        self.parser = xml.parsers.expat.ParserCreate()

        self.parser.pbuffer_text = True
        # don't break text content at line breaks

        # Install the three standard callbacks
        self.parser.pStartElementHandler = self.start_element
        self.parser.pEndElementHandler = self.end_element
        self.parser.pCharacterDataHandler = self.char_text

    def start_element(self, name, attrs):
        """Record name of start tag just encountered"""
        self.current_tag = name     # ignore attrs

    def end_element(self, name):
        """Stop parsing when species name has been encountered"""
        if name == 'BinomialOrgName_species':
            raise StopIteration

    def char_text(self, text):
        """If current tag is for genus or species, record its content"""
        if not text.isspace():
        # checking for whitespace because this function gets called
        # for both tag content and whitespace between the  end of
        # one tag and the beginning of the next
            if self.current_tag == 'BinomialOrgName_genus':
                self.genus = text.strip()
            elif self.current_tag == 'BinomialOrgName_species':
                self.species = text.strip()

    def find_binomial(self, filename):
        """Return (genus, species) as found in the XML genome file named filename"""

        # read only ChunkSize characters at a time in case we find it relatively early
        # and to not take up an enormous amount of memory with file contents
        with open(filename) as file:
            try:
                while True:
                    self.parser.Parse(file.read(self.ChunkSize))
            except StopIteration:
                pass
        return self.genus, self.species

if __name__ == '__main__':
    p = GenomeBinomialParser()
    print(p.find_binomial(sys.argv[1] if len(sys.argv) > 1
                          else 'data/XML/Example.xml'))
```

Getting the content of all tags with a specified name

In the XML file we are parsing, the names of the proteins are inside tags named `Seqdesc_title`. Example 8-18 shows a class that extracts a list of the names of all the proteins in the file.

Example 8-18. Getting content from all occurrences of a tag

```
import sys
import xml.parsers.expat

class GenomeProteinNameParser:

    def __init__(self):
        self.titleflag = False
        self.proteins = []                  # a form of collect iteration
        self.count = 0                      # just for user feedback
        self.parser = xml.parsers.expat.ParserCreate()
        self.parser.buffer_text = True
        # don't break text content at line breaks

        self.parser.StartElementHandler = self.start_element
        self.parser.EndElementHandler = self.end_element
        self.parser.CharacterDataHandler = self.char_data

    def start_element(self, name, attrs):
        """Keep a count of all Seqdesc_title tags and print a period to the terminal
        every 250 tags; if name == 'Seqdesc_title', turn on titleflag & ignore attrs"""
        if name is Seqdesc_title, ignoring attrs"""
        if name == 'Seqdesc_title':
            self.titleflag = True
            self.count += 1
            if not self.count % 250:
                print(self.count, file=sys.stderr)
                # sys.stderr so not buffered

    def end_element(self, name):
        """Set titleflag to False as soon as any end element is encountered"""
        self.titleflag = False              # regardless of close tag

    def char_data(self, text):
        """If last start tag was a Seqdesc_title, then add text to the list of protein names"""
        if self.titleflag:                  # turned on by start_element
            self.proteins.append(text.strip())

    def get_protein_names_from_file(self, filename):
        print('Parsing...')
        with open(filename, 'rb') as file:  # ParseFile requires bytes
            self.parser.ParseFile(file)     # parse entire file
        return self.proteins

def print_protein_names(filename, number):
    p = GenomeProteinNameParser()
    protein_names = p.get_protein_names_from_file(filename)
print('\n\nFound {} proteins; the first {} are:\n'.
```

```
        format(len(protein_names), number))
    for protein_name in protein_names[:10]:
        print(protein_name)

if __name__ == '__main__':
    default_filename = 'data/XML/AcidobacteriumCapsulatum.xml'
    print_protein_names(sys.argv[1] if len(sys.argv) > 1
                        else default_filename,
                        sys.argv[2] if len(sys.argv) > 2 else 10)
```

The output from running this program with no arguments is:

```
Parsing...
250 ...
500 ...
750 ...
1000 ...
1250 ...
1500 ...
1750 ...
2000 ...
2250 ...
2500 ...
2750 ...
3000 ...
3250 ...

Found 3383 proteins; the first 10 are:

Acidobacterium capsulatum ATCC 51196, complete genome.
cellulose synthase operon protein YhjQ [Acidobacterium capsulatum ATCC 51196]
hypothetical protein ACP_0957 [Acidobacterium capsulatum ATCC 51196]
hypothetical protein ACP_0614 [Acidobacterium capsulatum ATCC 51196]
hypothetical protein ACP_0540 [Acidobacterium capsulatum ATCC 51196]
putative membrane protein [Acidobacterium capsulatum ATCC 51196]
hypothetical protein ACP_0928 [Acidobacterium capsulatum ATCC 51196]
type I restriction-modification system, S subunit [Acidobacterium capsulatum
ATCC 51196]
putative flp pilus assembly protein CpaB [Acidobacterium capsulatum ATCC 51196]
glycosyl hydrolase, family 3 [Acidobacterium capsulatum ATCC 51196]
```

Tips, Traps, and Tracebacks

Tips

- You don't have to fit everything you are trying to do with HTML tags into a single regular expression. You can use several, or you can use one or more together with ordinary `str` methods.
- While developing a program, add functions for each step you recognize, even if you don't see anything for that step to do just yet. You can just use `pass` as the entire body of the definition, or just have it return its argument. Later, as you extend

the program, you will often find that the structure is already there and you'll just have to expand the definitions of one or more existing functions.

- Once you've got a program working with a set of web pages you want to process, make sure to try it on some similar web pages. Those experiments will often reveal cases you didn't handle in your regular expressions or function definitions, or bugs present in your program that weren't hit by the pages you were using initially.

Traps

- Don't just print the results of accessing the children of an `ElementTree` element—there could be thousands. First, assign a name to the result and check how many there are.

Tracebacks

Here are a few representative error messages:

`UnicodeDecodeError: 'ascii' codec can't decode byte 0xc2 in position 1784: ordinal not in range(128)`
: An attempt was made to read a Unicode character from a file opened with the default ASCII encoding rather than `encoding='utf-8'`.

`xml.parsers.expat.ExpatError: not well-formed (invalid token): line 1, column 1`
: Invalid XML appears at the beginning of a file; perhaps a blank line.

CHAPTER 9

Web Programming

This chapter introduces a variety of Python facilities for manipulating URLs, opening documents in web browsers, submitting HTTP requests to web servers, and executing programs on web servers to respond to HTTP requests. Along the way we will also look at the foundations of the socket technology that underlies web interactions and the basics of constructing HTML forms to use as an interface to server-based programs.

Manipulating URLs: urllib.parse

The `urllib.parse` module provides functions for manipulating URL strings. The general form of a URL is:

```
scheme://network_location/path;parameters?query#fragment
```

The *`fragment`*, which doesn't usually appear if the URL includes a *`query`*, is a reference to a particular place within a web page. You may not have seen or noticed the *`parameters`* portion of a URL before; it is not usually part of URLs visible to a browser's user, appearing—relatively rarely—as part of the value of `href` attributes in HTML tags. The *`network_location`* can be further dissected into the following components:

```
username:password@hostname:port
```

The *`username`*/*`password`* combination is a way of supplying login information to sites that accept this primitive kind of authentication. The *`port`* needs to be included whenever the server program that responds to a request for the given *`scheme`* is listening on a nonstandard port number. We'll see plenty of concrete examples of various forms of URLs in this chapter.

Certain characters are "reserved" for use in URL syntax and may not be part of the *`query`* component. These characters are represented by a percent sign (`%`) followed by their hexadecimal equivalent. You have undoubtedly noticed these in URLs, though you may not have realized what they were. Table 9-1 lists the characters and their URL equivalents. This translation is generally called *URL encoding*, but the official web standards call it *percent-encoding*.

Table 9-1. Reserved characters and their URL-encoded equivalents

!	*	"	'	(	)	;	:	@	&
%21	%2A	%22	%27	%28	%29	%3B	%3A	%40	%26
=	+	$	,	/	?	%	#	[	]
%3D	%2B	%24	%2C	%2F	%3F	%25	%23	%5B	%5D

In addition, for various reasons, spaces in are not allowed in URLs. Sometimes they are translated to plus signs (`+`) and sometimes to their hexadecimal equivalent, `%20`. Note that URL encoding is *not* the same as encoding of HTML entities such as `&` or, equivalently, `&`. Not only is the context completely different, but, to confuse things further, URL encoding uses hexadecimal numbers while HTML entities are encoded in decimal.

Disassembling URLs

Extracting a URL's components is a miniature parsing problem with enough complexity that Python provides library functions that implement it. You will want to use these instead of writing your own code:

`urllib.parse.urlparse(`*`string`*`[, `*`default_scheme`*`[, `*`fragmentflg`*`]])`

Returns an instance of `urllib.parse.ParseResult`, which looks like a six-element tuple whose elements correspond to the parts of the URL named in the preceding section. URL components not present in ***`string`*** are represented by empty strings, but if there is no scheme in ***`string`*** the tuple will contain ***`default_scheme`*** (if specified). If ***`fragmentflg`*** is false, a fragment in ***`string`*** is ignored.

Punctuation used to delimit fields, such as the `://` after the scheme name and the `?` before a query, are not included in the results. However, the initial slash of a path (if present) is retained, because in URLs that represent paths—those using a ***`file`*** scheme, in particular—an initial slash indicates an absolute path as opposed to a relative one (more on this later). In addition to its functioning as a six-element tuple the instance also has the read-only attributes shown in Table 9-2, the first six of which are shown in the order in which they appear in the tuple.

Table 9-2. Attributes of a ParseResult

Attribute	Description
`scheme`	URL scheme specifier
`netloc`	Network location
`path`	Hierarchical path
`params`	Parameters for last path element (uncommon)
`query`	Query component

Attribute	Description
`fragment`	Fragment identifier
`username`	Login information
`password`	Login information
`hostname`	The actual host without the login and port
`port`	Port number (integer) if present in location

`urllib.parse.unquote(`*`string`*`)`
: Returns a copy of `string` with all `%xx` sequences replaced by their one-character equivalents.

`urllib.parse.unquote_plus(`*`string`*`)`
: Just like `urllib.parse.unquote`, except that it replaces plus signs with spaces.

Assembling URLs

Programs that work with web technologies frequently construct URLs from details specified as parameters of programs and functions or fields of classes and instances. The `urllib.parse` module provides a full range of functions for constructing URLs:

`urllib.parse.urljoin(`*`base, url`*`[,` *`allow_fragments`*`])`
: Returns an absolute URL string constructed by adding to *`url`* any parts it is missing from the corresponding part of *`base`* (if present)

`urllib.parse.urlunparse(`*`parts`*`)`
: Returns a URL string constructed from *`parts`*, which is typically a `ParseResult` but could be any six-item iterable

`urllib.parse.urlencode(`*`data`*`)`
: Converts *`data`*, which is either a dictionary or a sequence of two-element tuples, to a URL-encoded string ready to be used as the argument portion of a query; the result is a string composed of *`key`*`=`*`value`* pairs separated by ampersands, with both *`key`* and *`value`* quoted by calling `urllib.parse.quote_plus`

`urllib.parse.quote(`*`string`*`)`
: Returns *`string`* with any special characters replaced by the corresponding `%xx` form to allow those characters to be included in a URL, especially in a query

`urllib.parse.quote_plus(`*`string`*`)`
: Just like `urllib.parse.quote`, except that it replaces spaces with plus signs instead of `%20`

`urllib.parse.urldefrag(`*`url`*`)`
: Returns a pair containing a copy of *`url`* without its fragment as its first element and its fragment as its second; the second element will be an empty string if *`url`* does not contain a fragment

Opening Web Pages: webbrowser

The `webbrowser` module provides simple mechanisms for displaying documents or query results in a web browser. The module's functions and methods all take a URL as their first argument and instruct the user's browser to display it.

> The functions in `webbrowser` give Python applications an easy way to use a browser as a presentation mechanism. An application can write some results into a text or HTML file to be displayed in the browser, submit a query through a browser, or just open a page of HTML-formatted documentation. Many other uses are also possible—all the program needs to do is provide a URL.

Module Functions

Following are descriptions of the `webbrowser` module's functions. All browser actions are subject to the browser in question's ability to perform those actions in the specified way with the user's current settings. For instance, if a function is described as opening a new tab, the actual action might be opening a new window. The functions are:

`webbrowser.open(`*`url`*`[, `*`new`*`=0[, `*`autoraise`*`=1]])`
: Displays `url` in a browser window (the current browser window if ***`new`*** is `0`, a new browser window if it's `1`, and a new browser tab if it's `2`); the window will be raised unless `autoraise` is false

`webbrowser.open_new(`*`url`*`)`
: Opens ***`url`*** in a new browser window (otherwise, it's opened in the current browser window)

`webbrowser.open_new_tab(`*`url`*`)`
: Opens ***`url`*** in a new browser tab (otherwise, it's opened in the current browser tab)

For example, to open the *python.org* home page:

```
webbrowser.open('http://www.python.org')
```

A function that opens the documentation for one of the core library modules follows:

```
def browse_module_doc(module_name):
    webbrowser.open('http://docs.python.org/py3k/library/' +
                    module_name + '.html')
```

Alternatively, you could replace `'http://docs.python.org'` with `'file://'` and the full path to where the documentation is in your Python installation (beginning with a slash, `/`, to make it absolute).

On some platforms, opening a URL may open it in some other kind of application. The type of URL—for example, *ftp* vs. *http*—may affect this behavior. If the URL is a path to a file (whether or not it begins with *file://*), the system might choose an application based on the file's extension or one associated with that specific file. For instance, you

may have configured your system to open all *.py* files in IDLE, in which case `webbrowser.open('/full/file/path/name.py')` may open *name.py* in IDLE. This makes the `webbrowser` module useful for many scenarios beyond those involving browsers.*

Constructing and Submitting Queries

An application can use `webbrowser.open` to submit queries like the ones you see in your browser's address bar when you submit a query through a website. The only difference between a query and a URL for an HTML document is that the query is a path to a program on the client that gets its arguments from the part of the URL after a question mark. For example, there's a package index at *http://pypi.python.org/pypi* that lists information about a large assortment of externally available Python software. If you were to type something into the search box at the top of the page and submit the query, the URL the browser would send to the *python.org* server would be of this form:

```
http://pypi.python.org/pypi?%3Aaction=search&term=text&submit=search
```

where *`text`* is whatever you typed in and `%3A` is the URL encoding for a semicolon. If your entry has more than one word, the individual words will be separated by plus signs. (In other words, the spaces in the multi-word query are replaced by plus signs, as described earlier.)

Example 9-1 shows a complete program for running a search of the Python Package Index from the command line.

Example 9-1. Searching the Python Package Index

```
"""Search the Python Package Index for one or more words"""

import sys
import webbrowser
import urllib.parse

urlpart1 = 'http://pypi.python.org/pypi?;action=search&term='
urlpart2 = '&submit=search'

def search_pypi(words):
    webbrowser.open(urlpart1 + urllib.parse.quote_plus(words) + urlpart2)

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: pypi term ...")
    else:
        search_pypi(sys.argv[1:])
```

* Technically, using `webbrowser` for *file://* URLs is unsupported and nonportable behavior, but if it does what you want, you might as well use it. Just make sure to note in your files that the behavior is undocumented and might well change in the future.

Constructing and Viewing an HTML Page

There are many ways to construct HTML pages. For instance, you could create an HTML document that includes `&` placeholders and use `string.Template` (see "String Utilities: string" on page 240 in Chapter 6) to replace them. (This is an ideal use for `string.Template` .) Another approach is to use `format` and assign some formatting strings for use in functions.

Example 9-2 shows an extension to the program for extracting links from HTML (shown in "Searching HTML text" on page 290). Assume you have run that program to produce some files that contain the pairs of URLs and text the program outputs. What this example does is use some of those files to produce a simple HTML file by putting each filename in an `h3` tag, making a new `a` tag for each URL, and enclosing the `a` tags in a numbered list. (There's a good chance that some of the links picked up by *extract_links* won't be meaningful or useful. In that case, you can always hand-edit the resulting file, or if there is a pattern to which links are useful and which aren't, you can filter them out either in that program's `get_all_tags` function or in this one's `make_html_for_file`.)

Example 9-2. Making a web page with links extracted from HTML files

```
import sys
import webbrowser

html_start = ('''<!DOCTYPE html PUBLIC
          "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>Collected Links</title>
</head>
<body>
''')

html_end = '''
</body>
</html>
'''

filetagformat = '\n<h3>{0}</h3>\n'
atagformat = '<li><a href="{0}">{1}</a></li>\n'

def make_html_file_from_files(filenames, outputfilename='links.html'):
    with open(outputfilename, 'w') as outfile:
        outfile.write(html_start)
        for filename in filenames:
            make_html_for_filename(filename, outfile)
        outfile.write(html_end)
    return outputfilename

def make_html_for_file(filename, outfile):
    outfile.write(filetagformat.format(filename))
```

```
    outfile.write('<ol>\n')
    with open(filename) as infile:
        for tag in get_all_atags(infile.read()):
            if tag[1]:
                make_html_for_tag(tag, outfile)
    outfile.write('</ol>\n')

def make_html_for_tag(tag, outfile):
    outfile.write(atagformat.format(tag[0], tag[1]))

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: browse_links filename ...")
    else:
        webbrowser.open(
            'file://' +
            os.path.abspath(make_html_file_from_files(sys.argv[:1]))
            )
```

Web Clients

The `webbrowser` module we looked at earlier provides a simple way to request the opening of a URL. Its functions return shortly after they are called, leaving a browser or some other application outside of Python to fulfill the request. In contrast, the `urllib.request` module provides a variety of sophisticated services for interacting with web servers from within Python itself.

The most generally useful functions in the module are:

`urllib.request.urlopen(`*`url`*`[, `*`data`*`[, `*`timeout`*`])`
: Performs the request specified by *`url`*, giving up after *`timeout`* seconds if specified; if a *`data`* argument is provided, it should be in the form returned by `urllib.parse.urlencode`, and it is supplied as the contents of a `POST` request (otherwise, the *`url`* itself would include the request's parameters); returns an object with the functionality of a file object and two additional methods:

    `urllib.request.geturl()`
    : Provides the URL of the actual resource retrieved; possibly redirected from the original request

    `urllib.request.info()`
    : Returns an instance of `http.client.HTTPMessage`, with information about headers, etc.

`urllib.request.urlretrieve(`*`url`*`[, `*`filename`*`[, `*`reportfn`*`[, `*`data`*`)`
: Performs the request specified by *`url`*, returning a two-element tuple containing the path to the file where the contents of the response was saved and the headers that would be returned by `urllib.request.urlopen(url).urllib.request.info()`; optional arguments are as follows:

filename
: The path to the file in which to save the query response; otherwise, a temporary file is used

reportfn
: If present, will be called when the network connection is established; then for each “block” of data read from the response, with three arguments: the number of blocks transferred so far, the number of bytes in a block, and the total size of the file

data
: If present, will be used as the contents of a POST request (and should be constructed by calling `url.parse.encode`); otherwise, query arguments must be provided as part of the URL itself

> Be careful using a string returned from `urllib.request.urlopen`, especially with regular expressions—the file-like behavior of the object returned provides a stream of bytes, not Unicode characters. Any read operation from the stream will therefore return a `bytes`, not a `str`. The simplest thing to do is just call `str` on the input, which will work if the bytes represent characters in the default encoding. You can specify an encoding by either giving it as the second argument to `str` or calling the `decode` method on the `bytes`, with the encoding as its argument.
>
> If you try to use a string for a regular expression to match a `bytes` target, you’ll get this error:
>
> ```
> TypeError: can't use a string pattern on a bytes-like object
> ```
>
> You could also use a `bytes` instead of a raw string for the regular expression, as long as the rest of the code is changed wherever necessary. The main point is that what go in and out over a network are bytes.

Making the URLs in a Response Absolute

URLs in a web page do not necessarily have a scheme (e.g., *http://*) or a hostname (e.g., *www.python.org*), making them pretty much useless as links. A URL without a scheme and host is known as a *relative URL*, because its actual address is derived from the site and path at which the document containing the link was found. If a program such as that shown in Example 9-2 is supposed to do something useful with these links, they must first be converted to full URLs so that they can be used outside of the context of the page on which they were found.

A relative address that begins with a slash (*/*) is treated as if it were prefixed with the same protocol and site as the page on which it is found. This is similar to how paths in filesystems are interpreted: an initial slash denotes the “root” directory, and any address that does not begin with a slash is considered relative to the directory that contains its document. If the address is just a filename, such as *other.html*, clicking on the link will look for that file in the directory in which the current document was found.

If there are any slashes inside the address, everything through the last slash is treated as a path starting with a subdirectory of the one containing the link's document. For example, a link *images/small/img1.jpg* in a web page at this URL:

```
http://www.example.com/top/middle/demo.html
```

would refer to a file at this location:

```
http://www.example.com/top/middle/images/small/img1.jpg
```

In contrast, because it begins with a slash, the link */images/small/img1.jpg* would instead refer to a file at this location:

```
http://www.example.com/images/small/img1.jpg
```

Expanding relative URLs to absolute ones could require a lot of code to cover all the possible permutations of which parts are present in a URL and which aren't. This would get complicated quickly. There's no need to bother with direct string manipulation, though, because the `urllib.parse` module described at the beginning of this chapter gives us the tools we need to disassemble and assemble URLs. The function `urllib.parse.urljoin` is all we need here. It's not even necessary to test whether a link's address begins with a scheme and a host before calling `urllib.parse.urljoin`—the function works with whatever parts of a URL are present in the first argument. Any parts present in both arguments get their values from the second argument.

One problem with the programs we have been using up to this point is that they process saved HTML files. Although they know the names of the files, they have no information about the original source of their content. What we'll do now instead of saving and then reading HTML files is use `urllib.request.urlopen` to read web pages directly. We won't even bother writing them to files. Then, when the pages are read, the code will know their sites and paths and be able to use them to adjust the links they contain. The command-line arguments will be URLs from which to read—HTML pages or requests. (A long list of URLs can be provided by command-line input file redirection.)

> Since web page contents are returned as `bytes` objects, not strings, if you want to write the modified contents to a file you'll have to either first convert them to strings or open the file in `'wb'` mode.

Constructing an HTML Page of Extracted Links

Let's add some code to Example 9-2 to make the URLs in the document absolute. That way, the files the program constructs will be usable on their own. Example 9-3 shows the functions that replace those in Example 9-2. Everything else stays the same, except that the `__main__` code calls `make_html_file_from_urls` instead of `make_html_file_from_files`. Also, the program must import `urllib.parse` and `urllib.request`.

Example 9-3. Constructing a web page directly from requests

```
import urllib.parse
import urllib.request

def make_html_file_from_urls(urls, outputfilename='links.html'):
    with open(outputfilename, 'w') as outfile:
        outfile.write(html_start)
        for url in urls:
            make_html_for_url(url, outfile)
        outfile.write(html_end)
    return outputfilename

def make_html_for_url(url, outfile):
    outfile.write(filetagformat.format(url))
    outfile.write('<ol>\n')
    for tag in get_all_atags(str(urllib.request.urlopen(url).read().decode())):
        if tag[1]:
            make_html_for_tag(url, tag, outfile)
    outfile.write('</ol>\n')

def make_html_for_tag(url, tag, outfile):
    outfile.write(atagformat.format(make_absolute(url, tag[0]), tag[1]))

def make_absolute(url, tagaddress):
    return urllib.parse.urljoin(url, tagaddress)
```

All the links in the HTML file produced by this revised program are absolute. You can open the resulting file in a browser and click on the links to go to their pages or submit their requests.

Downloading a Web Page's Linked Files

Occasionally, you may come across a web page that's full of links to files you would like to download. Some browsers, browser extensions, and applications provide tools for doing this. However, you might want to do this in a program to customize exactly which links are downloaded and where they get stored, or to more easily run the program as part of some automated task. It's easy to modify the code from the previous examples in this chapter for this purpose. We already know what the necessary steps are:

1. Parse command-line options.
2. Fetch a document or submit request.
3. Read from the request.
4. Convert the bytes read to a string.
5. Extract all links.
6. Save documents to a file.

There is one import twist we'll add to this program, though: we don't want to download *all* the links from a page, just those with certain extensions. We have no idea what kinds of documents, requests, media, and just plain useless things might appear on a web page, but we typically know what kinds of links we do want to download: perhaps all *.zip* and *.pdf* files, or all *.html* links. With that in mind, we'll set things up so that the command-line arguments for the program are a URL and one or more extensions to extract.

The program will not handle URLs with queries that are contained in the results returned by `urllib.request.urlopen`. Query URLs usually don't indicate the kinds of files they will obtain, and they usually return new HTML pages with their results rather than something you would want to save as a file. (Well, you might want to save them, but we've already handled various aspects of that, and you can always click on a request URL, save the file, and run this program, giving it a file URL with its path.)

First we'll look at the options the program will accept. The function in Example 9-4 sets up the option parser (see "Command-line options: optparse" on page 218), but it doesn't actually parse the options.

Example 9-4. Downloading links: setting up the option parser

```
def make_command_line_parser():
    optparser = optparse.OptionParser(
        usage='Usage: downloadLinks [--destdir dir] [-extension ext]* URL')
    optparser.add_option('-d', '--destdir',
                         help='directory to which files will be downloaded')
    optparser.add_option('-e', '--extension', action='append',
                         help='extension(s) of links to download')
    optparser.add_option('-l', '--list', action='store_true',
                         help="list, but don't download, links")
    optparser.set_defaults(extension=('zip', 'pdf'))
    return optparser
```

> One of the nice things about using `optparse` is that the details specified for the option parser do a good job of documenting the program's behavior for the programmer, in addition to providing help strings for its user.

Example 9-5 shows the "main" part of the program. It calls `make_command_line_parser` to get a parser, tells it to parse the options, then checks to see that there is only one argument other than the options. If the number of nonoption arguments is not 1, the program exits with the value `2`, the standard Unix value for command-line errors (as opposed to runtime errors). If there is exactly one argument, the action to be performed depends on whether the `list` option was specified. If it was, the program lists all the links it would download were the `list` not specified. Otherwise, it actually downloads them.

Example 9-5. Downloading links: main

```
if __name__ == '__main__':
    optparser = make_command_line_parser()
    (options, args) = optparser.parse_args()
    if len(args) != 1:
        optparser.print_help()
        sys.exit(2)                         # command-line error
    else:
        if options.list:
            list_links_from_url(args[0], options)
        else:
            download_links_from_url(args[0], options)
```

Regardless of whether the program lists or downloads links, it calls `get_url_links` to extract from the file any links whose extensions appear in the list indicated by the `extensions` option. That function calls `get_url_contents` to read the contents of the URL, then calls `extract_links_from_string`, which uses a regular expression to find all the relevant links. See Example 9-6.

Example 9-6. Downloading links: finding the links

```
def get_url_contents(url):
    response = urllib.request.urlopen(url)
    contents = response.read()
    response.close()
    return str(contents)

linkstring = r"'([^']+?\.(exts))'|\"([^\"]+?\.(exts))\""

def extract_links_from_string(string, extensions):
    results = re.findall(linkstring.replace('exts', '|'.join(extensions)),
                         string,
                         re.M | re.S | re.I)
    return [result[1] or result[2] for result in results]

def get_url_links(url, options):
    return extract_links_from_string(get_url_contents(url),
                                     options.extension)
```

The function `get_url_contents` implements the usual routine for reading the contents of an entire web page. Assuming link addresses are enclosed in either single or double quotes and checking the two cases separately, `extract_links_from_string` replaces the substring `'exts'` in the regular expression with a disjunction of the actual extensions. Only one of the two groups will match, depending on whether the address was enclosed in single or double quotes. In Chapter 8 we ignored this little problem, allowing single and double quotes to match either; here we fix that with a slightly more complex regular expression.

Listing the links just involves printing the results of `get_url_links`, and downloading the links is only a little more involved. Both are shown in Example 9-7.

Example 9-7. Downloading links: the real work

```
def list_links_from_url(url, options):
    for link in get_url_links(url, options):
        print(link)

def download_links_from_url(url, options):
    for n, link in enumerate(get_url_links(url, options)):
        path = urllib.parse.urlsplit(link).path
        targeturl = urllib.parse.urljoin(url, link)
        print(n+1, targeturl, sep='\t')        # show progress
        urllib.request.urlretrieve(targeturl,
                                   os.join(options.destdir,
                                           os.path.basename(path)))
```

The code in these examples constitutes the entire program.† Try it on a familiar web page to see what it does.

Web Servers

Web server technology is a mystery to many people. It conjures up images of heavy-duty hardware, special-purpose software, expert system administrators to keep things running, elaborate security precautions, and so on. Actually, though, even OS X and Windows XP can run what are called "personal web servers," and you can set them up with just a few clicks. (OS X uses the open source Apache web server that is the standard for Linux systems.)

Sockets and Servers

A web server is just one of many kind of services. You may already be aware of some others: your computer might set its time by contacting a time server; shared disks on your network are provided by file servers; and so on. In fact, time servers and file servers date back to the earliest computer networks, circa 1970. Another service you probably use is the Simple Mail Transport Protocol, which we covered in Chapter 6 (in "Sending email: smtplib" on page 224).

Such services are built on top of a low-level operating system construct called a *socket*. A socket is basically a pair of network streams managed by the operating system. A socket is identified by an address, the usual form of which is the combination of a hostname and a *port* number. The hostname can be an IP address such as 168.221.10.201, the name of a computer on the local area network (LAN), the name of an Internet host, and so on. The machine on which the program is running can be addressed with the empty string, `'localhost'`, or the IP address 127.0.0.1.

† A file containing all the code together is part of the downloadable code examples available from this book's website.

Ports with numbers below 1024 are reserved for specific services; for example, port 80 is reserved for web servers. Programs are in principle free to use port numbers from 1024 through 65535, but some ports in the range 1024 through 49151 have the status "registered"; this means that specific software products have reserved them for their own use, but unless those products are running in your program's environment no problems arise from using those port numbers.

> This little discussion of sockets and servers is here to prepare you to understand what's involved in implementing programs that run on a server and respond to user requests. You are not likely to find yourself programming at this low level, but it's not inconceivable: one use for a socket-based server would be to provide to other programs on the local area network data stored across many different sources (different databases, file directories, etc.) without those programs knowing where the data is found.

Server fundamentals

A server program listens on a special server socket for connection attempts. When a client program requests a connection, the server performs the following actions:

1. Create a new socket with a port number generated by the operating system.
2. Respond to the request by telling the client the port number.
3. Loop reading from the socket, processing the input, and writing a response out to the socket.

The procedure is similar on the client side:

1. Request a connection to a server, specifying the server's hostname and the port number on which it is listening for connection attempts.
2. Loop writing data to the socket, reading the response, and acting on the response.

Figure 9-1 illustrates the process.

Example 9-8 shows the server code and Example 9-9 the client code for a simple directory listing server. The client accepts input from the user, in the form of a filename that can include wildcard characters; it then sends that input to the server. The server performs a `glob.glob` (see "Filename Expansion: fnmatch and glob" on page 232) on the pattern and returns a string consisting of all the names of the files that matched, separated by newline characters. The client reads that string and prints it.

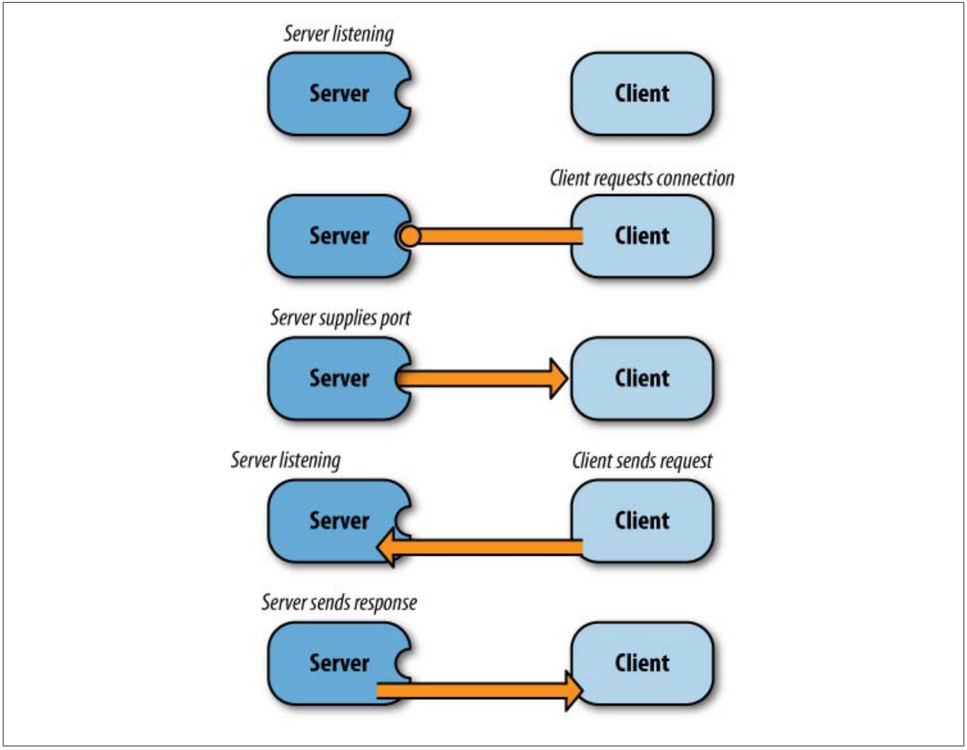


Figure 9-1. Client/server socket communication

Example 9-8. A directory listing socket server

```
import socket
import glob

HOST = socket.gethostname()                        # Instead of '', allows access over LAN
PORT = 5500                                        # Arbitrary nonprivileged port

# listener socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)    # standard
s.bind((HOST, PORT))
lisname = s.getsockname()
print('Listening on host', lisname[0], 'port', lisname[1])
s.listen(1)

# create actual socket
conn, addr = s.accept()
sockname = conn.getsockname()
print('Connection from', addr[0], 'on port', addr[1])

try:
    data = conn.recv(1024)
    while data:
        print(data)
```

```
        conn.sendall(b'\n'.join(glob.glob(data))l
        data = conn.recv(1024)
except KeyboardInterrupt:
    pass
finally:
    try:
        conn.close()
        s.close()
    except:
        pass
```

Example 9-9. A directory listing socket client

```
import socket

HOST = socket.gethostname()
PORT = 5500                                   # The server's listener port number

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)      # standard
s.connect((HOST, PORT))
sockname = s.getsockname()
print('Connected to', sockname[0], 'on port', sockname[1])

try:
    data = input('@ ')
    while data:
        s.send(data.encode(encoding))
        data = s.recv(1024).decode(encoding)
        print(data)
        data = input('@ ')
except (EOFError, KeyboardInterrupt):
    pass
finally:
    s.close()
```

A great many details and variations could be discussed; only the basics are demonstrated by these examples. In particular, there are important issues involved in managing multiple requests and larger amounts of transmitted data. For example, `socket.sendall`, used in the example's server code, keeps sending data until it has all been sent, even if it takes more than one network "packet" to transmit. The client code will have to take that into account and manage multiple packets.

> As with all network communications, this example reads and writes bytes, not strings. Since the function `glob.glob`, used to expand wildcards on the server side, accepts `bytes` arguments as well as strings, the server doesn't have to convert between bytes and strings. However, `socket.send` does require an argument of type `bytes`, so the client must encode the string read from the user. The response received could be printed without being decoded (as long as the default encoding works), but it just prints as one long string; after decoding, the `'\n'` characters that separate the filenames are printed as newlines.

Running a server: The http.server module

Organizations often have their web servers configured so that each user has a directory from which personal pages can be served, often called *html*. Even if yours does, though, you probably don't want your programming experiments visible throughout the entire organization. Python's `http.server` module provides an easy way to run a simple web server for your own purposes. (Others within your LAN will also be able to access it if you want to use it for more than just experiments.)

> Experimenting with running your own web server will help you better understand the fundamentals of the technology.

Example 9-10 shows the minimal setup code needed. This program will serve documents from the directory from which it was started. As it does, it will print information showing every request it gets from a client.

Example 9-10. Running a simple web server

```
import http.server
def run(port=8000,
        server_class=http.server.HTTPServer,
        handler_class=http.server.SimpleHTTPRequestHandler):
    httpd = server_class(('', port), handler_class)
    httpd.serve_forever()
run()
```

The function is parameterized for flexibility if used with other applications, but here we just call it with the default parameters. The tuple `('', port)` tells the server to listen on `port`. Since the official port assigned to the *http* protocol is 80, to access a web server that is listening on a port other than 80 you must include the port after the hostname in the URL, with a colon separating the two: *http://localhost:port*. This applies whether the server is running on your own computer, another computer on the LAN, or a computer somewhere on the Internet.

With the server running, enter *http://localhost:8000* in the address field of a browser. You should see a listing of the files contained in the server's directory. This is the default behavior of web servers not configured otherwise for directories that don't contain a file called *index.html* or *index.htm*. If one of those files is present, the browser will display that file rather than a directory listing. If a directory listing is displayed, clicking on one of the filenames will display it in the browser.

Unless the directory from which the server was started contains a file with the exact name *favicon.ico*, you will see repeated messages about not finding that file. A *favicon* is the tiny logo you see next to the URL in your browser's address field. Every time a browser accesses a document at a website, it also attempts to access that site's favicon.

A favicon is simply a 16×16 or 32×32 image in one of three formats: PNG, GIF, or ICO. The default name is *favicon.ico*, but a web page can specify a different filename and/or format (by means we won't go into here).

You might as well put one in your directory: that will stop the messages complaining about its absence and, even better, its presence in the browser will at least indicate that *something* is happening when you try to access a document even if no document appears. There's one on this book's website that you can use if you don't want to make your own. (It doesn't have to be anything special—a solid color is good enough; it just has to have one of the expected sizes and formats.)

Sometimes, especially during program development, the port the code tries to use will already be in use. This can happen because you've left a server running on that port in another window, or even because a program that aborted left the port open and the operating system didn't immediately close it. If you try the code in Example 9-10 several times in succession, interrupting it with Ctrl-Cs, you will probably see a `socket.error` message with the text `'Address already in use'`. That's the exception raised when the program fails in its attempt to open a socket that's already open.

To deal with this problem we can extend the code so that if there's an error trying to open a port it retries with the next port number, continuing to retry until it succeeds. We'll stop before ***port***`+100`, since if the code gets that far there's undoubtedly a worse problem than a conflicting port number. Example 9-11 shows the new version.

Example 9-11. Running a simple web server, finding a free port

```
def open_port(hostname, startport, server_class):
    for n in range(portnum, portnum+100):
        server_address = ('', portnum)
        try:
            return server_class(server_address, handler_class)
        except socket.error:
           pass

def run(portnum=8000,
        server_class=http.server.HTTPServer,
        handler_class=SimpleHTTPRequestHandler):
    httpd = open_port('', 8000, server_class)
    hostname, portnumber = httpd.socket.getsockname()
    print('Serving HTTP documents on {}, port {}...'
         .format(hostname, portnumber))
    httpd.serve_forever()
```

CGI

From the earliest days of web servers, requests submitted through web pages have used a mechanism called the *Common Gateway Interface* (CGI). There are more sophisticated mechanisms now, but CGI is still widely used. The CGI mechanism is nothing more than a standardized way of passing a URL request and its arguments to a program. After receiving the request, the program does its work and outputs HTML that gets sent back to the browser for display. The Python module that supports this mechanism is `cgi`.

As we saw at the beginning of this chapter, a URL often represents a request, rather than a specific document. The address part of a request identifies a program to be executed. A question mark identifies the beginning of the part of the URL containing the arguments of the request and their values. These arguments are the equivalent of keyword arguments in Python function calls, though they are not in any way specific to Python.

Serving CGI requests

A web server must be configured to execute CGI programs, or else a URL for a CGI program will simply display its text in the browser. Most web servers are configured to run CGI scripts. There are two ways they can distinguish CGI scripts from regular web pages: by their extension or by their location within the file hierarchy from which the server serves pages. Organizations that configure their web servers so that each user has a designated directory from which web pages may be served generally also configure things so that a subdirectory of that directory, generally called either *cgi-bin* or *cgi*, can contain CGI scripts.

If your organization has its web server configured appropriately, or you have a personal web server running on your computer that you can configure, you can use that server for experimenting with CGI scripts. In general, though, it is easier to do your experiments and development using Python's own simple server, as was shown in Example 9-10. The code in that example, however, only serves documents. To enable it to serve CGI requests too, the class `CGIHTTPRequestHandler` must be used in place of `SimpleHTTPRequestServer`, as shown in Example 9-12.

The default directories that contain programs `CGIHTTPRequestHandler` can run are listed in `CGIHTTPRequestHandler.cgi_directories`. The default value is `['/cgi-bin', '/htbin']`, but your program could change that (perhaps adding `'/cgi'` to the list, as is done in Example 9-12). These subdirectories are relative to the directory from which you start the server—if you start the server from */home/me/python/web*, that's what the server will interpret `/` to mean.

Example 9-12. Running a CGI server

```
http.server.CGIHTTPRequestHandler.cgi_directories.append('/cgi')
def open_port(hostname, startport, server_class):
```

```
    for n in range(portnum, portnum+80):
        server_address = ('', portnum)
        try:
            return server_class(server_address, handler_class)
        except socket.error:
           pass

def run(portnum=8000,
        server_class=http.server.HTTPServer,
        handler_class=http.server.CGIHTTPRequestHandler):
    httpd = open_port('', 8000, server_class)
    hostname, portnumber = httpd.socket.getsockname()
    print("Serving HTTP, including CGIs, on",
          hostname,
          "port",
          portnumber, "...")
    print('Serving HTTP documents and CGI requests on {}, port {}...'
         .format(hostname, portnumbe))
    print("CGI directories are:",
          http.server.SimpleHTTPRequestHandler.cgi_directories)
    httpd.serve_forever()
```

Documents can go in the same directory as the server, but executables must be in one of the subdirectories listed in the value of `CGIHTTPRequestHandler.cgi_directories`. Note that `CGIHTTPRequestHandler` can distinguish CGI scripts from documents only by their being located in one of these directories; it does not support extension-based determination the way full-powered web servers do.

Setting up CGI

When you first start working with CGI scripts, there are many mysterious things that can go wrong. Some of them result in completely blank web pages, with no hint of the nature of the problem. Sometimes what you get is a listing of the program instead of the result of its execution. Some of the problems are caused by various aspects of your CGI setup. Before trying to use your own code, you should take advantage of the `cgi` module's `test` function, which will send the browser a large amount of information about the CGI environment. Example 9-13 shows a complete CGI program using `cgi.test`. If `cgi.test` doesn't work, you haven't set things up entirely correctly.

Example 9-13. Running cgi.test

```
import cgi
cgi.test()
```

Run the program from Example 9-12 in a command window. It will run continuously until you interrupt it. Note the port it says it is listening on when it starts. Put the program shown in Example 9-13 in one of the subdirectories the program shows, naming it *cgitest.py*. On an operating system that supports "execute" permissions, change (with the `chmod` command-line command) the permissions of the file so that everyone can read and execute it (e.g., `755` on Unix-based systems).

Now, enter the following URL into your browser's address field, replacing ***port*** with the number of the port on which the server program is listening and ***cgi*** with the name of the directory in which you put *cgitest.py*:

```
http://localhost:port/cgi/cgitest.py
```

Your browser should display a long list of details of the environment in which the program ran.

CGI script arguments and responses

You won't learn much from `cgi.test`, except whether your CGI server environment is working and that you have your CGI script in the right directory with the right permissions. You will not see two lines that `cgi.test` prints first that are necessary to inform the browser that the program is sending it HTML to display. All CGI scripts must print these two lines before any other output:

```
Content-Type: text/html

<!-- the previous line must be blank; this line is just an HTML comment -->
```

Like many of the Python library's more frequently used modules, `cgi` provides several levels of interfaces. The simplest will suffice for most routine CGI scripting. The CGI Script template presented here shows the general outline of a CGI program. The program starts with the usual "shebang" line that specifies the program used to run the script. (This takes the place of typing `python3` and the name of the program at the command line.) To be run as a CGI script, the file must also be readable and executable by everyone, so set the file's permissions accordingly.

Because the shebang line specifies that this is a Python file, it does not need the *.py* extension to run. There's no need for users to be distracted by extensions like that, so it's good form to remove the extension from the filename. However, the extension is still needed so that operating systems, editors, and IDEs recognize it is a Python file when you open it for editing. The way around this conflict is to keep the *.py* extension on the file you are developing and create a link ("alias" or "shortcut") to it that does not have the extension. This also allows you to keep your files in a different directory than the one containing your CGI scripts—the link can be made from the file to your CGI directory.

When a CGI script is executed through a web server, everything it prints to `sys.stdout` is sent to the user's browser for display. (The newlines printed in the template are just for readability, for viewing a page's HTML "source" in a browser's "view source" window or the output when executing the program from the command line; in most contexts browsers ignore whitespace between the end of one HTML tag and the beginning of the next.)

TEMPLATE

Structure of a CGI Script

```
import cgi
import cgitb
cgitb.enable()

def respond():
    print(                    # the second line printed must be blank
'''Content-Type: text/html

<html>\n<head>\n<title>title</title>\n</head>\n<body>
''')
    args = cgi.FieldStorage()
    ... access args using getfirst and getlist ...
    print the HTML body
    print('</body>\n</html>')

if __name__ == '__main__':
    respond()
```

At the beginning the template initializes a very useful tool for debugging CGI scripts, which will normally be removed for a production script: importing and enabling `cgitb` arranges for nicely formatted runtime error reports to be displayed in the browser. It does not catch syntax errors, though—if your program has syntax errors, all you will see in the browser is a blank page. (You *will* see syntax errors reported in the window in which the server is running.)

The crux of Python's CGI module is the `FieldStorage` class. When an instance of the class is created, it parses the URL's query string and stores its arguments and values. Subsequently, the value of a query argument can be obtained by accessing the `FieldStorage` object as if it were a dictionary.

Because a request may have multiple values for the same argument, `FieldStorage` must support both single- and multiple-value arguments. (Even if your program expects just one value for an argument, a user might type in a URL that has more than one.) Dealing with this can get awkward, so `FieldStorage` provides two convenience methods: `getfirst(`*`argname`*`)` to return one value, and `getlist(`*`argname`*`)` to get a list of however many were present in the request. You can provide a second argument to `getfirst` (default `None`) that specifies the value to return if the request URL had no value for the argument. Normally it's better to obtain argument values this way rather than by accessing the values directly with dictionary notation.

The mechanism by which a web server passes arguments to a CGI script is really quite simple: all it does is set the environment variable `QUERY_STRING` to the part of the URL that follows the question mark. (There is another, more elaborate way of passing arguments that doesn't put them in the query string; getting the arguments in that case is a bit more complicated.) When a `FieldStorage` object is created, it checks whether

the os.environ dictionary contains the key 'QUERY_STRING'. If the key is present, its value is used to populate the FieldStorage object. If the environment does not contain 'QUERY_STRING', the FieldStorage object next looks for a command-line argument. If sys.argv[1] is present, it uses that as the query string.

> This means that you can test your CGI scripts by giving them a query string as a command-line argument. You don't need the whole URL, just the part after the question mark. This is especially useful for eliminating syntax errors before trying to run the script from a browser.

Let's fill in the CGI Script template with enough details to "echo" the arguments and values supplied in a URL. Example 9-14 shows what this would look like. In general, CGI scripts ignore arguments with unrecognized names; to demonstrate this usage, the program will only look for one argument named enzyme and any number of arguments named sequence.

Example 9-14. A CGI echo script

```
import cgi
import cgitb
cgitb.enable()

def respond():
    print("Content-Type: text/html")
    print()
    print("<html>\n<head>\n<title>title</title>\n</head>\n<body>")
    args = cgi.FieldStorage()

    # completing the template:
    print('<p>Enzyme:', args.getfirst('enzyme'), '</p>')
    print('<p>Sequences:', args.getlist('sequence'), '</p>')

    print("</body>\n</html>")

if __name__ == '__main__':
    respond()
```

If you have everything set up correctly, typing the following URL (together on one line, not with the line break):

```
http://localhost:8000/cgi/cgi_echo.py?enzyme='EcoRI'&
sequence=CCCC&sequence=CCGG&sequence=GGGG
```

into a browser should result in a web page that displays the following, unformatted:

```
Enzyme: 'EcoRI'

Sequences: ['CCCC', 'CCGG', 'GGGG']
```

Simple Web Applications

We'll close this section on web servers with a few real examples. The first is a straightforward application of the CGI techniques discussed earlier. The second will introduce some new information about creating web pages with simple forms and using those forms to submit requests to CGI scripts.

Using CGI scripts

This section's example is a CGI program to search Rebase data for all enzymes that recognize a particular site, specified as a `site` query argument in a URL. We'll assume that we have a datafile called *bionet.table* (bionet being one of the Rebase file formats), as written by the `write_table` function of Example 4-41 (see "Step 5" on page 159). Each line of that file contains the name of an enzyme, a tab, and a recognition sequence. Some of those recognition sequences contain a caret (^) indicating where the enzyme cuts the sequence, but we'll ignore those for this program.

Reading this file into a dictionary is simple, but there's a twist: we need to reverse the table. Instead of looking up recognition sites by enzyme, we want to find all the enzymes that recognize a particular site. Example 9-15 shows how to do this.

Example 9-15. Reading Rebase data into a reversed dictionary

```
def read_table(filename):
    table = {}
    with open(filename) as fil:
        for line in fil:
            enzyme, sequence = line.split()
            sequence = sequence.replace('^', '')      # ignore cut sites
            table.get(sequence, []).add(enzyme)
    return table
```

To generate the HTML for the CGI response, we are going to take advantage of the template facility in the `string` module (see "String Utilities: string" on page 240). This makes it easy to specify a large chunk of text that has only a few variable parts, which is just what we need (Example 9-16). In fact, we'll use a series of templates.

Example 9-16. The main template for the recognition site script

```
html_template = string.Template(                     # second line is requisite empty line
    '''Content-Type: text/html

<head>
<title>Restriction Enzyme Search</title>
</head>
<body>
<h2>Restriction Enzyme Search</h2>
$response
</body>
</html>
''')
```

There are three conditions the program must handle:

1. The URL did not include a `site` argument.
2. There are no enzymes that recognize the site.
3. There are one or more enzymes that recognize the site.

The program's responses for the three cases will be:

1. Display an error message.
2. Display a message that no matching enzymes were found.
3. Display an ordered (numbered) list showing the enzymes that recognize the specified site.

For each case the program will substitute appropriate HTML for the `$reponse` placeholder of `html_template` from Example 9-16. For the second and third cases the program uses templates to generate the response, as shown in Example 9-17; for the first case it just uses a plain string.

Example 9-17. Response Templates for the recognition site script

```
none_recognized_template = string.Template(
    '''<i>No enzymes recognize <b>$seq</b>.</i>\n''')

response_template = string.Template(
    '''<p>Enzyme(s) recognizing <b>$seq</b> are:
<ol>
$items
</ol>
''')
```

Example 9-18 shows the code that gets the script's one argument and produces the HTML to return to the browser. It initializes a dictionary for use with the main template, then for each of the three cases it sets the value of its `'response'` entry appropriately before performing a substitution.

Example 9-18. Code for the recognition site script

```
def make_html_body(table, seq):
    subst = {'seq': seq, 'items': [], 'response': ''}
    if not seq:                                    # case 1 -- no value for seq
        subst['response'] = 'No value for site in query arguments'
    else:
        seq = seq.upper()                          # doesn't change value in subst dict
        if seq not in table:                       # case 2 -- no match found
            subst['response'] = \
                none_recognized_template.substitute({'seq': seq})
        else:                                      # case 3 -- found, make list items
            items = '\n'.join(['<li>' + enzyme + '</li>'
                               for enzyme in table[seq]])
            subst['response'] = \
                response_template.substitute({'seq': seq, 'items': items})
    return subst
```

```
def print_response(table, seq):
    print(html_template.substitute(make_html_body(table, seq)))

def respond():
    print_response(read_table('data/Rebase/bionet.table'),
                   cgi.FieldStorage().getfirst('site'))

if __name__ == "__main__":
    respond()
```

As described earlier, you can test this script from the command line. For example:

```
% python3 enzymes_for_site.py site=cccggg
```

The output from this example should be HTML that lists three enzymes recognizing `'CCCGGG'`. (The script uses `string.upper` to make the sequence lookup case-insensitive.) If you get a runtime error complaining that the file *bionet.table* cannot be found, make sure that that file—or a link to it—is present in the directory from which the program runs, whether from the command line or through CGI.

> If you run the CGI program from a browser through a full-scale web server rather than using `http.CGIHTTPRequestHandler`, the datafile's permissions must be set to allow it to be "readable" by everyone, since scripts will not be run under your username.

HTML forms with CGI scripts

Although a URL for a request can be constructed programmatically, as in Example 9-1, requests are usually generated from web pages. HTML *forms* are the original method for doing this, and while old and simplistic, they are still a reasonable approach for simple web pages. More recent advances in client-side technologies are far beyond the scope of this book. The purpose of introducing HTML forms here is twofold:

1. It's easy to put together a simple interface with them, since the user's browser does most of the work.
2. It demonstrates the round-trip interactions involved in *client/server* technologies, of which web forms together with CGI scripts are an important example.

An HTML form is specified by a `form` tag. Any HTML tags and text can appear between the opening <form> and closing </form> tags. Certain tags designate form *controls*, which have fairly standardized appearance and behavior.

Generally, each control tag has a `name` attribute, which is used as the name of the parameter in any HTTP request the form submits. The `form` tag itself has an `action` attribute that specifies the URL of the program that will run when the form is submitted. (Frequently this is a relative URL—i.e., relative to the location of the page containing the form—though that is not a requirement.) When the request is submitted you will

see a new URL in your browser's address bar that includes the path to the CGI script, a question mark, and *`name=value`* pairs for each field in the form.

> Being able to see the request generated by submitting the form is a very helpful aid to debugging and learning more about CGI.

Client-side scripts (usually in JavaScript) may be attached to various events related to controls. Client-side scripts can also change the state of form controls. We will ignore client-side programming here and talk only about the default behavior of each kind of control and the interactions users have with them.

When a form is submitted, the value of a control is paired with the tag's `name` attribute to form the *`name=value`* pairs in the URL. The value is obtained from the state of the control in the browser—control tags do not need `value` attributes. However, if the form's HTML provides a `value` attribute for a control tag, that value will be used to initialize the state of that control. The kinds of tags that constitute forms are:

`button`
: There are three types of button controls, specified by the value of the tag's `type` attribute:

  `submit`
  : Clicking a submit button causes the form to be submitted—i.e., the names and values of the form's controls are combined with the form's `action` URL to create a request, which is then submitted to the web server (normally the server from which the form was obtained in the first place).

  `reset`
  : Clicking a reset button restores all the controls to their initial state.

  `push`
  : Push buttons have no default behavior; they are used together with scripts, so we won't be discussing them any further here.

`input`
: Input controls include the following, specified by the value of the tag's `type` attribute:

  `checkbox`
  : On/off switches that the user can toggle. Several checkboxes in a form may share the same `name` attribute value, allowing multiple values to be selected.

  `radio`
  : Like checkboxes, but they have a different appearance and normally share a `name` attribute value. The browser ensures that only one is selected at any time.

`text`
: A single-line area where the user can type text that becomes the value of the control.

`file`
: Opens a file selection dialog for uploading files.

`textarea`
: A multi-line input control (text box).

`select`
: A drop-down menu, each value of which is specified by an `option` tag.

Example 9-19 shows the HTML for a very simple web page. It has a form containing three elements: a label, an input box, and a submit button. The intent of the form is to generate a request to search the Rebase data loaded by the program shown in Examples 4-37 through 4-41.

Example 9-19. A simple HTML form for a restriction site query

```
<html>                                                  <!-- begin HTML -->
  <head>                                                <!-- begin HEAD -->
    <title>Restriction Enzyme Search </title>           <!-- window title -->
  </head>                                               <!-- end HEAD -->
  <body>                                                <!-- begin BODY -->
    <h2>Restriction Enzyme Search</h2>                  <!-- heading 2 -->
    <form action='cgi/enzymes_for_site'>                <!-- begin FORM -->
    <!-- action attribute specifies CGI script to run on submit -->
      <label>Recognition Site:</label>                  <!-- just text -->
      <input type='text' name='site' size='25'>         <!-- 25 char. input -->
      <button type='submit'>Find</button>               <!-- submit button -->
    </form>                                             <!-- end of FORM -->
  </body>                                               <!-- end of BODY -->
</html>                                                 <!-- end of HTML -->
```

Figure 9-2 shows how a browser would display the form shown in Example 9-19. (Of course, the actual fonts and possibly other details will differ according to which browser is used and how its preferences are set.)



Figure 9-2. The restriction site query form

Suppose the URL for the HTML page displayed in Figure 9-2 is:

```
http://localhost:8000/enzymes_for_site.html
```

If the user enters the sequence `CCGG` in the text input field and presses the submit button, the browser will submit the following request:

```
http://localhost:8000/cgi/enzymes_for_site?site=CCGG
```

The form's `action` attribute does not begin with an `http:`, so its path uses the same host as the HTML file. It doesn't begin with a slash, so the path is relative to the directory from which the server is running. Together that gives the partial URL `http://localhost:8000/`. The rest of the path, `cgi/enzymes_for_size`, is the value of the `submit` attribute of the `<form>` tag. When a form like this is submitted, the name and value of each of its input components is added to the URL, following a question mark. The name and value are separated by an equals sign, and if there are more than one the pairs are separated by ampersands.

The result will be something like what is shown in Figure 9-3.

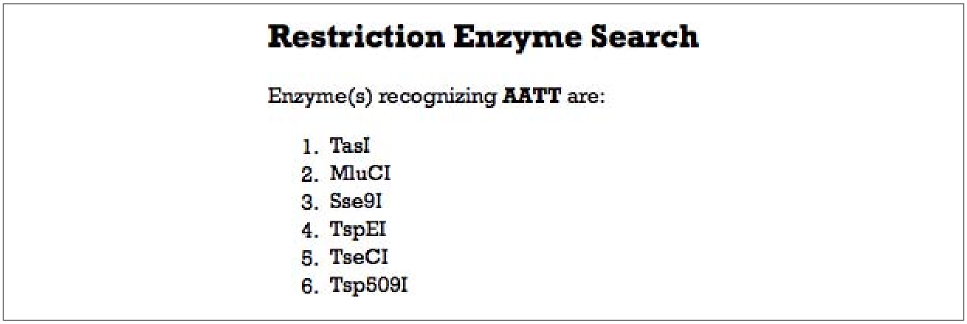


Figure 9-3. The restriction site query results

There's one more thing to take care of. The HTML produced by the preceding code does not include a form. Once the user has submitted one query, the web page returned can't be used to submit another one. True, the user could use the browser's back button, but for various reasons that is not a particularly good solution. It would be better to return the original form *plus* the results. All we'd need to do to implement this is extend the main HTML template to incorporate the HTML from the original form into the HTML for the response. Example 9-20 shows the new version of the HTML template originally defined in Example 9-16.

Example 9-20. Returning a form with CGI results

```
html_template = string.Template(
    '''Content-Type: text/html

<head>
  <title>Restriction Enzyme Search</title>
</head>
<body>
  <h2>Restriction Enzyme Search</h2>
    <form action='enzymes_for_site'>
      <label>Recognition Site:</label>
      <input type='text' name='site' size='25' value=$seq>
      <button type='submit'>Find</button>
```

```
    </form>
$response
  </body>
</html>
''')
```

Now, the user will see the results below the input area and will be able to use the form again. In addition, the previous entry in the input form will still be there, ready for editing in case the user wants to change it slightly. Figure 9-4 shows the results of the same query as Figure 9-3 after changing the HTML template as shown in Example 9-20.

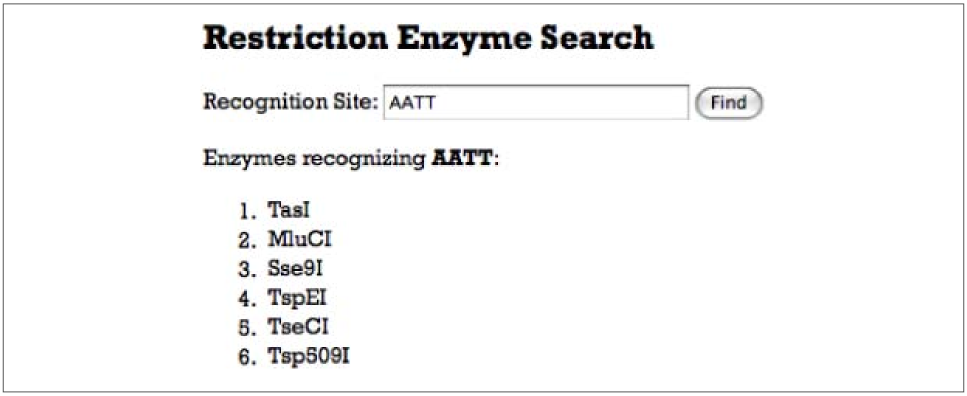


Figure 9-4. The restriction site query results with a form

We no longer even need the original HTML file now. We can just enter the query URL with no arguments to get the empty form:

```
http://localhost:8000/cgi/enzymes_for_site
```

> The `cgi/` component is not part of the `action` attribute of the generated HTML because the path will be relative to the directory where the CGI script that generated it is found.

Tips, Traps, and Tracebacks

Entrez Programming Utilities

The Entrez Programming Utilities (E-Utilities) provide uniform access to many of the Entrez databases. They are accessed through HTTP queries, but their parameters and responses are designed for programmatic use, as opposed to HTML forms and web pages marked up with formatting details. One of the query parameters specifies the desired form of output: text, XML, etc. The starting point for their documentation is *http://eutils.ncbi.nlm.nih.gov/entrez/query/static/eutils_help.html*. That page provides

links to the documentation of each of the tools and a short course describing how to use them to construct data pipelines (*http://www.ncbi.nlm.nih.gov/bookshelf/br.fcgi?book=coursework&part=eutils*). Perhaps the most useful query in the context of the kinds of data access shown in this book is `Efetch`, specifically in its use with "Sequence and other Molecular Biology Databases," as described at *http://eutils.ncbi.nlm.nih.gov/corehtml/query/static/efetchseq_help.html*. A complete list of the software tools available from the NCBI website is at *http://www.ncbi.nlm.nih.gov/guide/data-software*.

Tips

Web pages

- Put the following function, which we saw earlier, in a file of Python utilities you maintain, and import it into the files you are working on. (For a module like this it does make sense to use the `from ... import *` form of the `import` statement.) You can call this function from within Python to open the documentation of a given module. You'll probably want to give it a shorter name, but there's no need to change the name of the function itself. Instead, simply assign the name you want to `browse_module_doc`:

```
def browse_module_doc(module_name):
    webbrowser.open('http://docs.python.org/3.1/library/' +
                    module_name + '.html')

mdoc = browse_module_doc

>>> mdoc('re')             # browse re module doc
>>> mdoc('os.path')        # browse os.path module doc
>>> mdoc('modules')        # browse modules index
>>> mdoc('index')          # browse library index
```

- Like Python strings, HTML attribute values may use single or double quotes. Mixtures of single and double quotes can cause a great deal of confusion and frustration when you are constructing HTML strings, matching HTML text with regular expressions, or using ordinary string facilities to take apart HTML. If your program is constructing HTML, you can choose which quotes to use where, and you should make a point of using either single quotes for Python strings and double quotes for HTML tag attributes, or vice versa. If your program is analyzing HTML obtained from elsewhere, and that HTML uses either single or double quotes consistently, use the other kind for your strings. Of course, if you always use triple quotes in these situations—even for one-line strings—you will avoid such problems entirely.
- It is very tricky to develop and debug both web interactions and program actions together. Whenever you do any kind of web programming, develop the code first using data from a locally stored file. (In particular, if the data is supposed to be obtained by a query, perform the web query manually and save the results to a text file.) Once your code is working, add the actual web interaction.

Debugging CGI scripts

- In general, you can run your CGI script from the command line, giving it a single argument that contains arguments the way they would appear in a query following its question mark: in the form of `name=value` pairs separated by ampersands. This can be very helpful for debugging. (Usually you'll have to quote the argument on the command line, especially when there is more than one value, since most command-line shells treat ampersands specially.)
- Importing your script into the interpreter and running pieces of it manually is another effective debugging technique.
- Always import and include `cgitb.enable` in your code so that you get its elaborate runtime error messages while developing CGI scripts.
- Syntax errors in CGI will show up in the window from which you are running one of the `http.server` module's servers.

HTTP servers

- The `http.server` module's servers print every request they get to the window in which they are running.
- You can stop the favicon messages by installing a *favicon.ico* file (see "Running a server: The http.server module" on page 341) in the directories from which you run web servers (Python's or otherwise).
- Use links (aliases) to *.py* CGI files, omitting the file extension. This is a form of information hiding. (This may not work in Windows environments, where the extension is the only way to tell the system what program to use to execute a file.) Keep the following points in mind:
  - —You want the URL submitted to look like a query, not a filename. The implementation of the query as a Python script is a detail that should not be exposed to the outside world. (Don't you get a bit irritated or confused when you see *.php* or *.pl* or even *.cgi* extensions in URLs?)
  - —Because the alias separates the query name from the actual file that implements it, you can change the path to the actual file and re-create the alias or link to the new path without having to change anything in the HTML files that use the query in form actions.
- Give your queries (the aliases or links to your *.py* files, or the *.py* files themselves if you aren't using aliases or links) descriptive but not enormously long names. Users may read the query URLs in their browsers' address bars, bookmark them, etc. Later, they may search their history or bookmarks for a URL and should be able to find it by recalling a natural syllable in the query name.
- Likewise, keep the paths of your query aliases relatively short, even if the actual files are buried deep in your file hierarchy.

- When you're running an `http.SimpleHTTPServer` or `http.CGIHTTPServer` from a command-line Python program, syntax errors will appear in the window from which it was started. Tracebacks will not. In order to view tracebacks you must include the following lines in your program:

  ```
  import cgitb
  cgitb.enable()
  ```

  Those tracebacks will appear (in dramatic color) in the web browser. Obviously, you would not want to enable this when people are actually using your CGI script.

Traps

- *All network programming protocols return* `bytes` *objects*. You must call the `decode` method on `bytes` objects you want converted to strings. You can specify the encoding to use as an argument; otherwise, the default decoding (usually the local variant of ASCII) will be used.
- Web servers execute CGI scripts as separate executables with no shell environment. Therefore, if using a "shebang" line it must specify the absolute path to the Python executable, rather than using */usr/bin/env*.
- For a file to be executed as a CGI script using an `http.server` module server:
  - —You must use `http.CGIHTTPServer`: it both serves documents and executes CGI scripts, whereas `http.CGIHTTPServer` only serves documents.
  - —The script file must be executable.
  - —The `http.CGIHTTPServer` recognizes a request as a CGI script simply by checking the path from the directory in which the server was started to the directory of the script. Extensions (or lack thereof) have no effect. The subdirectory can be several levels down.
- Absolute and relative URLs in an HTML file are treated very differently. Absolute URLs begin with a slash: their paths begin at the directory from which the web server is running. Relative URLs do not have a leading slash; their paths begin at the directory from which the web page was obtained.
- Running scripts through a full-featured web server rather than using one from Python's `http.server` module introduces further constraints. The exact details depend on the site's configuration, but the following are likely to be issues. Many are the result of the server's executing scripts under a special username reserved for the server. The considerations in this list were much more of an issue in earlier days of organization web servers, but you should be aware of them, especially if your organization may be using default configurations that retain some of these restrictions and requirements:
  - —Scripts must be readable and executable by "anyone."
  - —Datafiles accessed by scripts must be readable by "anyone."

— Directories on the path from a script's directory to any datafile it uses must be navigable (probably readable and executable) by "anyone."
— Whether a file is interpreted as a script to execute may depend on the name of the directory in which it is found, its extension, or a combination of the two. (If only certain extensions are allowed, you may have to use a *.cgi* extension in the alias or links to your actual *.py* files.)
— It may be necessary to place scripts and HTML files in designated directories under your home directory—*html*, for example.
— Your scripts may not be accessible from the Internet, depending on the site's configuration, firewall settings, etc.
— Links or aliases from within your designated HTML and CGI subdirectories to files in directories outside of those directory trees may not be allowed, or they may work for HTML but not for CGI.

Tracebacks

Here are some error messages you are likely to encounter while doing web programming with Python's libraries:

`socket.error: Address already in use`
: An attempt has been made to start a server on a port that is already being used; typically, this will happen when a server you started has not stopped and released its port—either kill the process manually or start a new server using a different port number.

`TypeError: can't use a string pattern on a bytes-like object`
: You matched a regular expression string of type `str` against bytes returned from `urllib.request.urlopen`; convert the bytes to a string by calling `str`, with an *`encoding`* argument if necessary, or call the `decode` method on the bytes, giving it an *`encoding`* argument if necessary.

`urllib.error.HTTPError 400: Bad Request`
: An error has been returned from a web server in response to a call to `urllib.request.open` or the like.

`urllib.error.URLError: <urlopen error [Errno 8] nodename nor servname provided, or not known>`
: In a call to `urllib.request.open`, the URL didn't contain a hostname, or your computer is not connected to the Internet.

CHAPTER 10

Relational Databases

Most generally, a *database* is simply an organized collection of data. The data can be stored in memory, in files in one of your directories, in publicly accessible files, or in files on a computer acting as a *database server*. Like a web server, a database server stands between client programs and the actual data, so that applications never directly access the files storing the data.

There isn't necessarily anything special about database files, other than the structure given to their data by the applications that access them. Spreadsheets have long been used informally for recording and manipulating data in files. Desktop office application suites generally include a database component that offers capabilities beyond those of spreadsheets. These database components allow you to access data selectively instead of having to read and write entire spreadsheet documents. They also provide mechanisms like pattern-based searching that can be used to construct desktop applications.

Database servers, in turn, are far more powerful and can hold far more data than either spreadsheets or desktop office suite database components. You may have heard of commercial database systems such as Sybase and Oracle and open source systems such as PostgreSQL and MySQL. Database servers use highly specialized file formats that support capabilities such as concurrent access by multiple users, restrictions on access to data, a variety of efficiency techniques, automatic backup, and a wide range of other features necessary for group- and enterprise-scale data management. Client applications access data on these servers through a standardized language known as the *Structured Query Language* (SQL).

Most database server technology is based on a *relational data model*. The theory of relational data models is a rather complex subject. Fortunately, we can use the less formal terminology of SQL, the standardized language used with relational databases.* This section presents a very basic introduction to the relational model and SQL. Next, we'll explore an example of designing and loading data into a database.

* As an acronym for Structured Query Language, "SQL" should be pronounced "ess cue ell," though many people pronounce it like the word "sequel."

Once we have data in a database, we can look at some examples of database queries. The chapter culminates with the development of a web page and a CGI script that allow the database to be queried through a form in a browser.

Representation in Relational Databases

A full introduction to relational database design would more comfortably occupy an entire book than a section of a chapter. The goal here is to give you enough information to get you started using relational databases. We will cover the essentials and demonstrate them with an example.

Database Tables

A relational database stores data in *tables* that, like spreadsheets, are organized into rows and columns. Each row represents an individual *entity* (a neutral term used to avoid committing to a particular interpretation of what the data in the row means). Each column represents an attribute of the entities in the tables. In Python terms, a table is like a class (type), a row is like an object, and a column is like an attribute, with each column of a row holding the value of the corresponding attribute.

As in Python dictionaries, the entries in relational database tables are identified by *keys*. A key for a table's entries may comprise the value of a single column or the values of several columns. Because a table's key identifies an individual row, each row's key must be distinct from the keys of every other row in that table.

A table may have more than one key, since there may be more than one column or combination of columns that reliably distinguishes each row from every other. Usually, however, one of the keys is designated as the *primary key*. This is not a requirement of the relational data model, but a pragmatic aspect of the way database systems define and manage tables. Database systems also offer a mechanism by which an integer-valued column can be designated as a primary key with values automatically generated in sequence.

Avoiding duplication of values

One important principle of good database design is that combinations of column values should not appear in more than one row of a table. To give a minimal example, suppose we write down the four ways to refer to an amino acid, as shown in Table 10-1. This violates the rule, because each combination of values for Code, Short, and Name occurs in more than one row.

Table 10-1. A minimal example of column dependency

Codon	Code	Short	Name
UUU	F	Phe	Phenylalanine
UUC	F	Phe	Phenylalanine
UUA	L	Leu	Leucine
UUG	L	Leu	Leucine
UCU	S	Ser	Serine
UCC	S	Ser	Serine
UCA	S	Ser	Serine
UCG	S	Ser	Serine
. . .	. . .	. . .	. . .

Bad things happen when this rule is violated. For example, "phenylalanine" may be mistakenly spelled differently in the two rows in which it appears. That would mean that "UUU" and "UUC" would refer to different amino acid names even though they refer to the same amino acid code and short names, creating a serious inconsistency in the data. Another problem is that repeating combinations of column values causes the table to take up more space than is necessary.

The way to fix these kinds of problems is to create a second table to contain just a single copy of the repetitive information, as shown in Table 10-2. Any of the three columns in this table can be used as a key, because given the value of one of the columns, the values of the other two are uniquely defined.

Table 10-2. A database table for amino acid codes and names

Code	Short	Name
F	Phe	Phenylalanine
L	Leu	Leucine
S	Ser	Serine
. . .	. . .	. . .

Table 10-1 can now be revised by dropping two of the redundant columns, as shown in Table 10-3.

Table 10-3. A revised codon-amino acid table

Codon	Code
UUU	F
UUC	F
UUA	L
UUG	L

Codon	Code
UCU	S
UCC	S
UCA	S
UCG	S
. . .	. . .

Given these two tables, here are the steps for obtaining the full name of an amino acid given a codon:

1. The row with the specified codon as the value of its Codon column is located in the codons table.
2. The value of the Code column of that row is used as a key to the amino acids table.
3. Finding the row with that value in its Code column, the value of the Name column is obtained.

In the vocabulary of dictionaries, the value of an entry in the first dictionary is used as the key to access a value in the second dictionary. The technical term for one or more columns of one table that are used to look up a row in another table is a *foreign key*—foreign because it's a key in a different table.

Representation of relationships

Together, Tables 10-2 and 10-3 define a relationship between codons and amino acids. This is called a *one-many* or *1-N relationship*, because for every one entry in the amino acid table there are potentially many (N) entries in the codon table. Occasionally a database will have a *one-one* or *1-1 relationship*, where each entry in one table corresponds to at most one entry in the other. More common than 1-1, but less common than 1-N, are relationships in which for each entry in one table there can be many in the other, and vice versa. This kind of relationship is called a *many-many* or *M-N relationship*.

> Strange as it may sound, the term "relationship" is not what the word "relational" refers to in the terms "relational database" or "relational data model." "Relation" is a mathematical term akin to "function," in the sense of mapping a domain into a range. "Relationship" is a design and implementation term signifying an interconnection among objects.

M-N relationships require special handling. Consider a table representing researchers (Table 10-4) and another representing institutions (Table 10-5). The primary key for the Researcher table will be the researcher's Social Security number or equivalent ID. In the Institution table we'll make the simplifying assumption that no two institutions have the same name, so we can use the name as the primary key. (If two institutions

did have the same name, we could use a combination of the institution's name and state as the primary key, or we could use something more definitive, such as the institution's government tax ID.)

Table 10-4. A table of information about researchers

ID	First	MI	Last	Position	Institution
111-22-3333	Blue	J	Bird	Professor	Some Univ.
444-55-6666	Kay	C	Jones	Student	Another Univ.
777-88-9999	Elm		Oh	Postdoc	Some Univ.

Table 10-5. A table of information about institutions

Name	Street	City	State	Country
Some Univ.	1 A St.	Oneville	MA	USA
Another Univ.	20 B St.	Twotown	NY	USA
Corporate R&D	9 Ave. Q	Oneville	MA	USA

So far we have the same kind of 1-N relationship we had in the previous example: N researchers for any institution, one institution for any researcher. Now let's make Blue J. Bird a consultant at Corporate R&D, as shown in Table 10-6.

Table 10-6. Adding another entry for a researcher

ID	First	MI	Last	Position	Institution
111-22-3333	Blue	J	Bird	Professor	Some Univ.
444-55-6666	Kay	C	Jones	Student	Another Univ.
777-88-9999	Elm		Oh	Postdoc	Some Univ.
111-22-3333	Blue	J	Bird	Consultant	Corporate R&D

Now we have a problem: rows 1 and 4 have the same primary key—the ID—which violates the requirement that no two rows have the same value for a key. The problem here is that an M-N relationship is essentially two 1-N relationships: one researcher can have multiple positions and, of course, one institution can have multiple researchers. However, in a 1-N relationship only one of the tables can have rows that repeat values for the foreign key to the other table.

M-N relationships must be represented in a separate table whose rows contain foreign keys to each of the other two tables. First we'll create a table called "Appointment." We'll add to it a column for the foreign key from each of the two tables involved in the relationship. The primary key for this table will be the combination of the values in those columns. Table 10-7 illustrates this initial step. Note that no two rows have the same value for the combination of researcher and institution.

Table 10-7. Creating an M-N Appointment table

Researcher	Institution
111-22-3333	Some Univ.
444-55-6666	Another Univ.
777-88-9999	Some Univ.
111-22-3333	Corporate R&D

Next, we'll remove the foreign key column from the Researcher table, as well as the columns that differ between the rows with the same ID. We'll then add those columns to the new table. In this example the only column other than the foreign key is the Position column. Moving the Position column to the Appointment table makes sense, because a position is associated with the combination of researcher and institution: a person isn't a professor, but rather a professor at a particular institution. Table 10-8 shows the modified Appointment table and Table 10-9 the modified Researcher table.

Table 10-8. Adding a column to the Appointment table

Researcher	Institution	Position
111-22-3333	Some Univ.	Professor
444-55-6666	Another Univ.	Student
777-88-9999	Some Univ.	Postdoc
111-22-3333	Corporate R&D	Consultant

Table 10-9. Adding a column to the Researcher table

Researcher	First	MI	Last
111-22-3333	Blue	J	Bird
444-55-6666	Kay	C	Jones
777-88-9999	Elm		Oh

Database definition

The table definitions of a relational database constitute its *schema*. Many formal notations and diagramming techniques exist for specifying complex schemas, but for simple databases it is often sufficient just to list the table details in text format. Specifically:

- The names of the database's tables
- The name of each column of each table
- The type of value contained in each column of each table
- For each column of each table, whether its values can be null
- The column(s) of each table forming the key by which its entries can be referenced

- The primary key of each table
- The columns of each table having values that are keys in another table (the foreign keys)

A database containing both the amino acid tables and the researcher tables could be described textually as shown in Table 10-10. (The asterisks denote primary keys; the type `char` represents a fixed number of characters, while `varchar` represents a variable number of characters up to the specified limit.)

Table 10-10. An informal database schema

Table	Column	Type	Null?	Foreign key
AminoAcid	*Code	char(1)	N	
	Short	char(3)	N	
	Name	char(13)	N	
Codon	*Codon	char(3)	N	
	Code	char(1)	N	AminoAcid.Code
Researcher	*ID	integer	N	
	First	varchar(255)	N	
	MI	char(3)		
	Last	varchar(255)	N	
Institution	*Name	varchar(255)	N	
	Street	varchar(255)		
	City	varchar(255)	N	
	State	char(3)	N	
	Country	char(5)		
Appointment	*ID	text	N	Researcher.ID
	*Institution	text	N	Institution.Name
	Position	text		

A Restriction Enzyme Database

In this section we're going to build a small database that holds the information about restriction enzymes contained in *http://rebase.neb.com/rebase/link_allenz* ("allenz" standing for "all enzymes"). This version of the Rebase data provides substantially more information than the version we used in our earlier examples. We'll begin with the

structure of the datafile, a program to read that data, and a design for the database based on it. We'll assume that the file is saved under the name *allenz.data* in the same directory that will contain the Python file with the code to process it.

The data

After some introductory text, which we'll skip as we've done in a number of other examples, the datafile contains an entry for each enzyme in the following format:

```
<1> Enzyme Name
<2> Prototype
<3> Microorganism
<4> Source
<5> Recognition Sequence
<6> Methylation Site
<7> Commercial Availability
<8> References
... blank line ...
```

The details of each field are as follows (conveniently, the numbers in angle brackets are actually part of the datafile format):

1. The *enzyme name* is the standard identifier of the enzyme (e.g., EcoRI).
2. The *prototype* is just the name of another of the file's enzymes, which was the first one discovered that recognizes the same sequence as the entry's enzyme. Entries for prototype enzymes have an empty prototype field.
3. The *microorganism* field consists of two or more subfields that name the *genus*, *species*, and possibly *subspecies* information. Some subspecies entries consist of somewhat informal multiword descriptions; others contain a single well-known subspecies designation.
4. The *source* is the name of an individual or National Culture Collection.
5. The *recognition sequence* is a coded description of both the sequence recognized by the enzyme and the way it cuts at that site.
6. The *methylation site* describes the location within the recognition sequence methylated by the cognate methylase; only about 10% of the enzymes in the file have information in this field. We will ignore this field.
7. *Commercial availability* is a series of single-letter codes indicating which vendors supply the enzyme. We will ignore this field.
8. As stated at the beginning of the datafile from which the rest of this information was extracted, "only the primary references for the isolation and/or purification of the restriction enzyme or methylase, the determination of the recognition sequence and cleavage site or the methylation specificity are given." The information in the *references* field is expressed as a list of one or more integers separated by a comma and a space. The integers refer to numbered references whose details are listed after the entry for the last enzyme.

The content of the recognition site field is a series of DNA bases, written for one strand only, from 5' to 3'. The cleavage site—if known—is represented by a caret (^), placed within the base sequence so that the cleavage occurs on the 3' side of the caret.

Some enzymes cleave several bases away from an end of the recognition site. For these, the base sequence in the field is followed by an indication of the location of the cut site(s). This is expressed with parentheses and slashes: for example, `(5/10)`. The first number in this example indicates that the 5' to 3' cleavage occurs 5 bases downstream from the end of the recognition site, and although the complementary sequence is not shown, the second number indicates that the cleavage occurs 10 bases before the recognition site on the 5' to 3' complementary strand. Some enzymes cut on both sides of their recognition sequences, in which case the field will contain a parenthesized pair of numbers before and after the recognition sequence.

Reading the data

As we've seen in earlier examples, files often begin and/or end with information that is not part of the information we want to extract from them. The datafile we are working with in this case actually has two data sections, so the steps that skip, read, and stop at nondata must be performed twice. We'll read the data using an implementation of the Grand Unified Bioinformatics File Parser (see "The Grand Unified Bioinformatics File Parser" on page 146). The code is shown in Example 10-1.

Example 10-1. Reading Rebase "all enzyme" data

```
def read_enzymes_from_file(filename):
    with open(filename) as file:
        skip_intro(file)
        return (get_enzymes(file), get_references(file))

def skip_intro(file):
    """Skip through the documentation that appears at the beginning
    of file, leaving it positioned at the first line of the first enzyme"""
    line = ''
    while not line.startswith('<REFERENCES>'):
        line = file.readline()
    while len(line) > 1:                        # always 1 for '\n'
        line = file.readline()
    return line

def get_enzymes(src):
    enzymes = {}
    enzyme = next_enzyme(src)
    while enzyme:
        enzymes[enzyme[0]] = enzyme             # dict key is enzyme's name
        enzyme = next_enzyme(src)
    return enzymes

def read_field(file):
    return file.readline()[3:-1]
```

```
def read_other_fields(file):
    """The name of the enzyme has already been read;
    read the rest of the fields, returning a 7-tuple"""
    return [read_field(file) for n in range(7)]         # read 7 fields

def next_enzyme(file):
    """Read the data for the next enzyme, returning a list of the
    form: [enzyme_name, prototype, source, recognition_tuple,
    (genus, species, subspecies), references_tuple]"""
    name = read_field(file)
    if name:                                             # otherwise last enzyme read
        fields = [name] + read_other_fields(file)
        fields[2] = parse_organism(fields[2])
        fields[7] = [int(num) for num in fields[7].split(',')]
        file.readline()                                  # skip blank line
        return fields

def parse_organism(org):
    """Parse the organism details"""
    parts = org.split(' ')
    if len(parts) == 2:
        parts.append(None)
    elif len(parts) > 3:
        parts[2:] = ' '.join(parts[2:])
    return tuple(parts)

def skip_reference_heading(file):
    """Skip lines until the first reference"""
    line = file.readline()
    while not line.startswith('References:'):
        line = file.readline()
    file.readline()                                      # skip following blank line

def next_reference(file):
    """Return tuple (refnum, reftext) or, if no more, (None, None)"""
    line = file.readline()
    if len(line) < 2:                                    # end of file or blank line
        return (None, None)
    else:
        return (int(line[:4]), line[7:-1])

def get_references(file):
    """Return a dictionary of (refnum, reftext) items from the
    references section of file"""

    # using a dictionary here because there are many references
    # and an enzyme may reference several of them
    refs = {}
    skip_reference_heading(file)
    refnum, ref = next_reference(file)
    while refnum:
        refs[refnum] = ref
        refnum, ref = next_reference(file)
    return refs
```

```
if __name__ == "__main__":
    if len(sys.argv) < 2:
        filename = 'data/Rebase/allenz.data'
    elif len(sys.argv == 2):
        filename = sys.argv[1]
    else:
        print('Usage: read_enzymes [filename]')
    enzymes, references = read_enzymes_from_file(filename)
    print('Read', len(enzymes), 'enzymes and',          # print len instead of
          len(references), 'references')                # contents of huge lists
```

A schema for the Rebase database

There are many graphical conventions for representing database schema, but we don't need anything fancy. Figure 10-1 should suffice. What it shows is:

- The name of each table
- The names of the columns of each table
- A 1-N relationship from Enzyme to Organism, meaning there is only one Organism for each Enzyme, but there may be more than one Enzyme for each Organism
- A 1-N relationship from the Enzyme to itself, meaning there is (optionally) one prototype Enzyme for each Enzyme, but a single Enzyme can be the prototype of many others.
- A M-N relationship, represented in a separate table, between the Enzyme and Reference tables

The 1-N relationships are drawn the way they are because the relationship is represented in the database by a column in the N-side table that contains the key of the 1-side table.

Table 10-11 is a textual representation of the diagram in Figure 10-1.

Table 10-11. A textual schema for the Rebase data

Table	Column	Type	Null?	Foreign key
Organism	*OrgID	integer	N	
	Genus	varchar(255)	N	
	Species	varchar(255)	N	
	Subspecies	varchar(255)		
Reference	*RefID	integer	N	
	Details	varchar(255)	N	
Enzyme	*Name	varchar(255)	N	
	Prototype	varchar(255)		Enzyme.Name[a]

Table	Column	Type	Null?	Foreign key
	OrgID	integer	N	Organism.OrgID
	Source	varchar(255)	N	
	RecognitionSeq	varchar(255)	N	
	TopCutPos	integer		
	BottomCutPos	integer		
	TopCutPos2	integer		
	BottomCutPos2	integer		
EnzymeReference	*Enzyme	varchar(255)	N	Enzyme.Name
	*RefID	varchar(255)	N	Organism.OrgID

[a] Enzyme.name is a key back to the same table, not a foreign key, but it serves the same purpose—the establishment of a 1-N relationship. There is nothing strange about one row in a table being related to another.

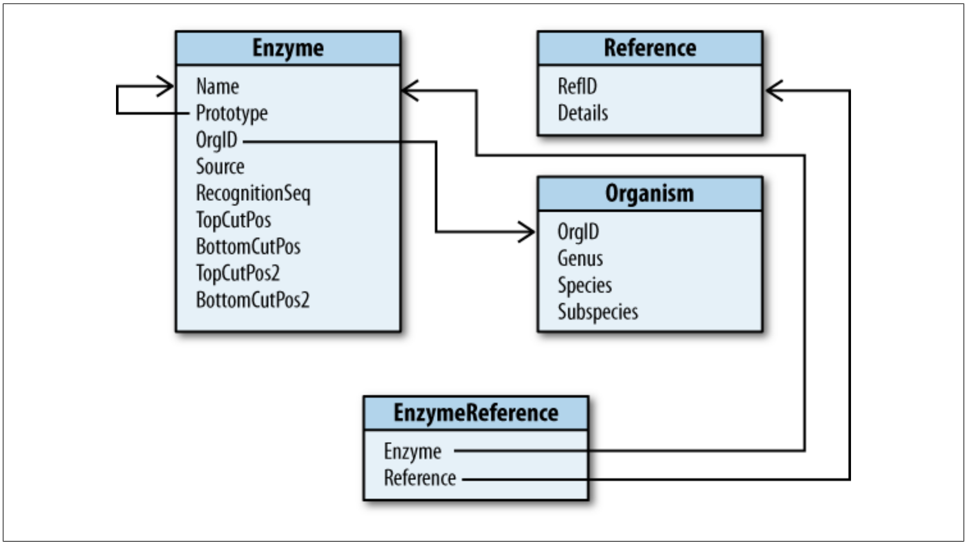


Figure 10-1. Schema diagram for the Rebase example

Using Relational Data

Given the preceding introduction and example, we can now look at the ways database tables and their contents are manipulated in practice.

SQL Basics

There are three major kinds of SQL operations:

Modify the schema
: Add, remove, or modify tables.

Modify table contents
: Add, remove, modify, or replace rows of tables.

Query the database
: Get results matching database queries.

These operations are performed using SQL statements, as shown in Table 10-12.

Table 10-12. SQL statements

Category	Statement	Operation
Modify database schema	`CREATE TABLE`	Add table
	`DROP TABLE`	Remove table
	`ALTER TABLE ... RENAME TO`	Rename table
	`ALTER TABLE ... ADD COLUMN`	Add a column
Modify table contents	`INSERT INTO`	Add row(s)
	`DELETE`	Remove row(s)
	`UPDATE`	Modify a row(s)
	`REPLACE`	Replace a row(s)
Query database	`SELECT`	Obtain results

The rest of this chapter describes SQL fundamentals and how SQL is used from Python. As it proceeds, it uses an extended narrated example. Certainly much more could be done with this example, just as much more could be explained about SQL. The material here is intended to be consistent in depth with the material in the rest of this book.

Using the sqlite3 module

Large-scale database systems provide interactive interpreters, graphical user interfaces, and programs for executing SQL statements contained in text files. All of that is beyond the scope of this book and even of much of the work programmers do to use a database. What's important for programming purposes is that database systems provide libraries that implement functions (or methods, if object-oriented) for other code to call. We will use the SQLite3 database[†] and Python's **`sqlite3`** module that interfaces to it because it is easy to use, it does not require a separate server, and it is included in the Python installation.

† You can find further information about SQLite3 at *http://www.sqlite.org*.

Python defines an abstract interface to SQL database systems. Accessing a specific database system requires a module that implements the interface for that kind of database. Implementations of the interface for many server-based systems can be found through the Python Package Index.‡

The `sqlite3` module is an interface to a SQLite3 library compiled into the Python interpreter, so it doesn't require SQLite3 to be installed on your computer. However, if SQLite3 *has* been installed there will be a command-line interpreter program you can run called *sqlite3*.§

If you use the command-line *sqlite3* program, bear in mind that standard interactive SQL tools require each statement to end with a semicolon.

While SQL is supposedly a standardized language, many database systems extend it in various ways and/or fail to implement some of its features. It is therefore always somewhat risky to show how to do things for relational databases in general, since a given SQL statement might not work with a specific database system. The SQLite3 database system is intentionally quite minimalist, but it nevertheless implements a large portion of SQL in a fairly standard way. Even though SQLite3 does not use a database server, it provides the same sort of interface used by full-featured client/server systems.

You'll want to learn more about databases and the `sqlite3` module to accomplish your goals, and perhaps learn about other database systems as well. Beyond the pragmatics of actually using one, this section should give you a fairly clear picture of how relational databases manage and manipulate data. Given that foundation, it shouldn't be too difficult to learn about other uses and other database systems.

Connecting to the database

The first step in using the `sqlite3` module is to create a database *connection* object:

```
conn = sqlite3.connect(dbname)
```

where ***dbname*** is the name of the file containing the database. The `sqlite3` module provides convenient high-level methods for connection objects that hide some lower-level details that are part of interacting with a database connection. We'll look at three of them here:

‡ Located at *http://www.python.org/pypi*; it is unclear how long it will be before each of these is updated to run with Python 3.

§ If SQLite3 has not been installed on your machine and you want to try installing it, go to *http://www.sqlite.org/download.html*. There are several versions available for download. The one you want (as of this writing) is *sqlite3-3.6.14.bin.gz* for Linux, *sqlite3-3.6.14-osx-x86.bin.gz* for Intel Macs running 10.5+, or *sqlite-3_6_14.zip* for Windows. The numbers after the first "3" will vary according to release.

connection.execute("*a SQL statement*"[, *values*])
: Executes the SQL statement, replacing placeholders in the statement (discussed momentarily) with the corresponding element of ***values*** (a sequence or dictionary). Results (if any) are returned as an iterable that produces tuples containing the values for one result.

connection.executemany("*a SQL statement*", *values-sequence*)
: Executes the SQL statement once for each sequence of values in ***values-sequence***, replacing placeholders in the statement with the corresponding values, as with ***connection*.execute**. This is primarily used for insertions of multiple rows into tables.

connection.executescript("*a SQL script*")
: Executes multiple SQL statements. This is a convenient way to create all the tables of a database in one method call, as you'll see later in Example 10-4. Note that each statement must end with a semicolon.

Programmatic SQL interfaces provide mechanisms for parameter substitution into statements. (This is one of those areas that is not as standardized as it should be.) The way it works in `sqlite3` is very simple. The second argument to **`execute`** or **`executemany`** can be either a sequence or a dictionary. The placeholder used with a sequence is a question mark; when the query is executed, each question mark is replaced by the corresponding element of the sequence. There must be exactly as many elements in the sequence as there are question marks in the query. If the values are supplied in a dictionary, the query contains keys of the dictionary preceded by a colon, and the dictionary must contain every key used in the query (though it may contain others as well). We'll see examples of both mechanisms later.

The `sqlite3` module defines an exception class, `sqlite3.Error`. You should always embed calls to **`execute`** inside **`try`** statements that catch this error and print out information about what has happened. After printing the error the exception clause should reraise the error with **`raise`**, since it won't normally be able to fix the problem.

Changes SQL statements make to a database are accumulated without actually being made to the underlying file. To move the changes into the file, it is necessary to *commit* them. This is the rough equivalent of closing an output file. The SQL **`COMMIT`** statement serves the same purpose in all standard SQL tools.

`Connection` objects provide the following methods for managing their state:

connection.commit()
: Writes the accumulated changes to the database

connection.abort()
: Abandons the accumulated changes without writing them to the database

close()
: Closes the connection *without committing the changes*

The accumulation of changes before a commit or rollback is called a *transaction*. Transactions are a critical part of programming large-scale databases. They prevent incomplete sets of changes from being made when an error occurs. For example, a banking system must ensure that money withdrawn from one account is deposited into another, so it uses a transaction to ensure that either both changes occur or neither does. Also, databases manage transactions so that one program's changes don't interleave with another's—the banking system shouldn't update the balance based on what was in the account when its transaction started, because that amount may have been changed by another transaction in the meantime.

> A useful feature of methods of `sqlite3` module classes is that they implicitly open a transaction whenever a SQL statement that changes the database schema is encountered. They also commit the implicit transaction when the next statement that does not change the schema is executed. This avoids schema changes and content changes interfering with each other.

Creating the database

SQL statements to create the four tables of the Rebase database look something like those shown in Example 10-2. (Many variations are possible even for a particular SQL implementation.) SQL keywords are usually shown in uppercase, so we'll follow that convention here.

Example 10-2. Database table creation statements for Rebase data

```
CREATE TABLE Organism(
    OrgID integer PRIMARY KEY,
    Genus text NOT NULL,
    Species text NOT NULL,
    Subspecies text
    );

CREATE TABLE Reference(
    RefID integer PRIMARY KEY,
    Details text NOT NULL
    );

CREATE TABLE Enzyme(
    Name text PRIMARY KEY,
    Prototype text,
    OrgID integer NOT NULL REFERENCES Organism(OrgID),
    Source text NOT NULL,
    RecogSeq text NOT NULL,
    TopCutPos integer,
    BottomCutPos integer,
    TopCutPos2 integer,
    BottomCutPos2 integer,
    );
```

```
CREATE TABLE EnzymeReference(
    Enzyme text NOT NULL REFERENCES Enzyme(Name),
    RefID integer NOT NULL REFERENCES Reference(RefID),
    PRIMARY KEY(Enzyme, RefID)
    );
```

The beginning of each of these statements names a table. Then, for each of the table's columns, the statement lists its name, type, and zero or more *constraints*. Constraints on the table itself follow the list of columns. Basic column constraints include:

`NOT NULL`
: A value must be assigned to the column when a row is inserted or changed.

`PRIMARY KEY`
: The column is the table's primary key and, implicitly, `NOT NULL`.

`UNIQUE`
: No two rows in the table may have the same value for the constraint's column.

`DEFAULT` *`value`*
: This specifies the value to give to a column in newly created rows when no value is specified for that column.

`REFERENCES` *`TableName`*`(`*`ColumnName`*`, ...)`
: The column is a foreign key corresponding to the column(s) of the same name in the specified table.

Constraints that may appear on the table itself include:

`PRIMARY KEY(`*`ColumnName`*`[, ...])`
: An alternate form of `PRIMARY KEY` constraint; if the primary key involves more than one column it must be specified as a constraint on the table.

`FOREIGN KEY(`*`ColumnName`*`[, ...]) REFERENCES` *`TableName`*`(`*`ColumnName`*`, ...)`
: An alternate form of `FOREIGN KEY` constraint; if the foreign key involves more than one column it must be specified as a constraint on the table.

The next template shows the general form of the SQL statement for creating a table.

SQL

Creating a Table

The `CREATE TABLE` statement gives the name of the table to create, followed by a list of column specifications and other information. With brackets indicating optional components and ellipses repetition, the basic form of SQL's table creation statement is:

```
CREATE TABLE table_name(column_name column_constraint ...,
                        column_name column_constraint ...,
                        ...
                        [table_constraint [, table_constraint
                         ... ]]
                        )
```

If you are redefining a table and reloading its data, you should drop the table first. The statements shown in Example 10-3 would normally precede the creation statements in Example 10-2, allowing changes to be made to a program's table creation statements before it is reexecuted.

Example 10-3. Dropping tables from the Rebase database

```
DROP TABLE IF EXISTS Organism
DROP TABLE IF EXISTS Reference
DROP TABLE IF EXISTS Enzyme
DROP TABLE IF EXISTS EnzymeReference
```

Dropping a table removes both the table's contents and its definition from the database. The only variation on the `DROP TABLE` statement is that the `IF EXISTS` portion may be omitted. An attempt made to drop a nonexistent table without `IF EXISTS` raises an exception.

SQL

Dropping a Table

The `DROP TABLE` statement removes a table from the database. Normally you would include `IF EXISTS` because an exception will be raised if you try to drop a table that is not in the database.

```
DROP TABLE [IF EXISTS] tablename
```

The `sqlite3` Python code to execute the drop and create statements to define our example database is shown in Example 10-4. We use the nonstandard convenience method `Connection.executescript` to drop and create all the tables in one call, and a closing `commit` to tell the database to accept all the changes.

Example 10-4. sqlite3 code to create the Rebase database

```
import sqlite3
conn = sqlite3.connect(datafilename)
try:
    conn.executescript('''
DROP TABLE IF EXISTS Organism
DROP TABLE IF EXISTS Reference
DROP TABLE IF EXISTS Enzyme
DROP TABLE IF EXISTS EnzymeReference

CREATE TABLE Organism(
    OrgID integer PRIMARY KEY,
    Genus text NOT NULL,
    Species text NOT NULL,
    Subspecies text
    );
```

```
CREATE TABLE Reference(
    RefID integer PRIMARY KEY,
    Details text NOT NULL
    );

CREATE TABLE Enzyme(
    Name text PRIMARY KEY,
    Prototype text,
    OrgID integer NOT NULL REFERENCES Organism(OrgID),
    Source text NOT NULL,
    RecogSeq text NOT NULL,
    TopCutPos integer,
    BottomCutPos integer,
    TopCutPos2 integer,
    BottomCutPos2 integer,
    );

CREATE TABLE EnzymeReference(
    Enzyme text NOT NULL REFERENCES Enzyme(Name),
    RefID integer NOT NULL REFERENCES Reference(RefID),
    PRIMARY KEY(Enzyme, RefID)
    );
    ''')

    except: sqlite3.OperationalError as err:
        print(err, file=sys.stderr)
        conn.rollback()              # abort changes
        raise
conn.commit()                        # commit changes
```

Tables can also be renamed, as shown in the next template.

SQL

Renaming a Table

A table can be renamed as follows:

```
ALTER TABLE tablename RENAME TO new_name
```

Internal database mechanisms depend on column names and types in a number of important ways that make changing the name, type, or constraints of a column problematic or even impossible. Database systems differ in the details. Other than renaming, the only kind of modification to a table's definition that SQLite3 supports is adding a column.

SQL

Adding a Column

To add a column to a SQLite3 database:

```
ALTER TABLE tablename ADD COLUMN columnspec
```

The column specification is the same as in a `CREATE TABLE` statement, except that certain kinds of constraints may not be included.

Loading data into tables

The statement used to add a row to a table is `INSERT INTO`. Only a few of the insertion statements for our example database are shown, because altogether it will have approximately 15,000 rows!

SQL

Inserting a Row into a Table

The basic SQL statement for inserting a row into a table has the form:

```
INSERT INTO tablename VALUES(value1, value2, ..., valuen)
```

where `n` must be the number of columns in the table. The values must appear in the order of the corresponding columns.

When a program is doing insertions, a better approach is to use a question mark in place of each value and supply the actual values as the statement is repetitively executed:

```
connection.execute("INSERT INTO tablename
                    VALUES(?,?,?,...)",
                    value1, value2, ..., valuen)
```

As always, a dictionary may be used to supply the values instead of a sequence, with names preceded by colons in place of the question marks.

The first step in loading the data into the database is to read all the information from the datafile, as shown in Example 10-1. What that produces is a dictionary of enzyme information keyed by the enzyme names, and a dictionary of references keyed by the numbers as they appear in the datafile. The Reference table can be loaded directly from the references dictionary, but the other three tables require some manipulation of the raw data: we have to split the enzyme data into the information that goes into the Organism table and the information that goes into the Enzyme table, and we need to create the M-N table that connects the Enzyme and Reference tables.

Example 10-5 shows the code for loading the data from the text file into the database. The top-level function is `load_data`. It calls a separate function for each table:

- `load_reference_data`
- `load_organism_data`
- `load_enzyme_data`
- `load_enzyme_reference_data`

In addition, `load_enzyme_data` calls `recognition_info` to parse the cut site string into its components. We're ignoring that detail here but leaving the function in as a "stub" that just returns a list of its sequence argument and four zeros. We will deal with these details at a later point.

Example 10-5. Loading the Rebase database

```
def load_data(dbname, enzymes, references):
    """Reorganize the information in the enzymes and references dictionaries in accordance
    with the dbname's schema and load the data into the database's tables"""
    try:
        conn = get_connection(dbname)
        load_reference_data(conn, references)
        organism_ids = load_organism_data(conn, enzymes)
        load_enzyme_data(conn, enzymes, organism_ids)
        load_enzyme_reference_data(conn, enzymes)
        conn.commit()
    except sqlite3.OperationalError as ex:
        print(ex, file=sys.stderr)
        raise             # reraise exception so Python can handle it

def load_reference_data(conn, references):
    for refid, ref in references.items():
        store_data(conn, 'Reference', (refid, ref))

def load_organism_data(conn, enzyme_data):
    """Return a 'reverse' dictionary keyed by the tuples in the list enzyme_data, which have
    the form (Genus, Species, Subspecies); the dictionary's values are sequentially generated
    integer IDs, which will be used as foreign keys in the Enzyme table"""
    organism_ids = {}
    for orgid, data in enumerate(enzyme_data.values()):
        # generating OrgIDs as we go
        org = data[2]
        # this is the only part of an enzyme's data that is relevant here
        if not org in organism_ids:
            store_data(conn, 'Organism', (orgid+1,) + org)
            organism_ids[org] = orgid+1
    return organism_ids

def load_enzyme_data(conn, enzymes, organism_ids):
    for data in enzymes.values():
        store_data(conn,
                   'Enzyme',
                   (data[0],                        # name
                    data[1] or None,                # prototype (None if '')
                    organism_ids[data[2]],          # organism ID
                    data[3]) +                      # source
                   recognition_info(data[4]) +      # recognition sequence
                   ('',)                            # methylation site
                    )

def load_enzyme_reference_data(conn, enzymes):
    """For each reference of each element of enzymes add an entry to the M-N EnzymeReference
    table with the name of the enzyme and the ID of the reference"""
    for data in enzymes.values():
```

```
        for refid in data[7]:                        # refid list
            store_data(conn, 'EnzymeReference', (data[0], refid))

# temporary implementation: does nothing
def recognition_info(seqdata):
    """Parse the recognition sequence data seqdata, returning a tuple of the form:
        (sequence, top_cut_pos, bottom_cut_pos, top_cut_pos2, bottom_cut_pos2)"""
    return (seq, 0, 0, 0, 0)
```

These four functions make use of the code shown in Example 10-6 to avoid lengthy program text and code repetition. The function `make_insert_string` generates an `INSERT` statement with the right number of question marks for a table of a given name and number of columns. `STORE_STMTS` is a dictionary keyed by table name that contains the result of `make_insert_string` for each of the four tables of the database. The function `store_data` encapsulates the execution of an `INSERT` statement inside a `try` statement. It obtains the appropriate string for the `INSERT` statement from the `STORE_STMTS` dictionary.

Example 10-6. Utilities to support Rebase database loading

```
def make_insert_string(tablename, n):
    """Return an INSERT statement for tablename with n columns"""
    return ('INSERT INTO ' + tablename + ' VALUES ' +
            '(' + ', '.join('?' * n) + ')')     # n comma-separated ?s

STORE_STMTS = {tablename: make_insert_string(tablename, ncols)
               for tablename, ncols in
               (('Organism', 4),
                ('Reference', 2),
                ('Enzyme', 9),
                ('EnzymeReference', 2))}

def store_data(conn, tablename, data):
    """Store data into tablename using connection conn"""
    try:
        conn.execute(STORE_STMTS[tablename], data)
    except Exception as ex:
        print(ex)
        raise
```

SQL Queries

With tables defined and data loaded into the database, it is time to consider the crux of SQL use: queries that fetch information from the database. This final part of the chapter explores the key aspects of SQL data access. The topic is far too complex for us to cover anything but the most basic techniques, but the ones we will cover account for a significant proportion of what even a moderately experienced SQL programmer would use in practice. Also keep in mind that implementations in other databases may vary in some minor ways from the SQLite3 syntax described here.

Simple database queries

Creating and loading tables is essentially just administrative work. The real power—and complexity—of SQL comes into play in accessing the information that's already stored in a database. A key advantage SQL has over directly accessing a program's data structures is that it provides many powerful mechanisms for choosing columns, combining data from multiple tables, imposing filters, performing calculations, and constructing temporary tables that can be used as the source for further accesses.

The heart of SQL is its `SELECT` statement. A `SELECT` statement specifies combinations of column names, table names, calculations, conditions, and so on used to refer to data. The execution of a `SELECT` statement produces a (possibly empty) stream of results from which the program can read.

SQL

Simple SELECT Statement

The simplest form of the `SELECT` statement is:

```
SELECT column_name, ... FROM table_name
```

The result is a stream of tuples containing values in the designated columns for every row of the specified table. To access all the columns of a table, use the shorthand:

```
SELECT * FROM table_name
```

In documentation and discussion of `SELECT` statements, each part after the column names—`FROM` ***`table_name`***, for instance—is referred to as a *clause*. `SELECT` statements can have a variety of optional clauses. An important one in practice is `LIMIT`, since for tables with many rows, such as those in our Rebase example, you probably won't want to access all of them.

SQL

Limiting SELECT Statement Results

You can limit the number of rows a query returns as follows:

```
SELECT column_name, ... FROM table_name LIMIT number
```

You can also add another clause to `SELECT` statements that have a `LIMIT` clause to say where in the sequence of results you want to start (the "offset"):

```
SELECT column_name, ... FROM table_name LIMIT number OFFSET position
```

Another option is to add a clause specifying the column(s) to use in determining the order of the results:

```
SELECT column_name, ... FROM table_name ORDER BY column_name
```

Column names are not the only thing that can follow the SELECT keyword. A wide range of expression types can be used, with syntax much like Python's. (An important exception is that a single equals sign is an equality test; there is no assignment.)

SQL

Expressions in SELECT Statements

Some of the more important expressions are:

```
column_name
literal_value
expression(s)
expression1 binary_operator expression2
expresssion IS/NOT NULL
expression1 [NOT] LIKE expression2
expression1 [NOT] BETWEEN expression2 AND expression3
function_name(expression, ...)
```

("Literal" values are numbers, single-quoted strings, NULL, etc.) SQL functions you can use in SELECT statements include COUNT, MAX, MIN, AVG, and SUM. An expression following LIKE includes wildcard characters, with _ matching a single character and % zero or more. Matching is case-insensitive.

Another maneuver that is often important is to limit expressions—COUNT(), in particular—to distinct values. This is like the difference between a set and a tuple, in that sets do not contain duplicate values.

SQL

DISTINCT in SELECT Statements

The SQL keyword DISTINCT can appear before the expressions of a SELECT statement or the arguments to a function (inside the parentheses). This qualifies the query so that only one row for each distinct value is included in the results.

```
SELECT COUNT(DISTINCT Genus, Species) FROM Organism
```

Example 10-7 shows some simple SELECT statements for the Rebase database. Example 10-8 shows code to execute them in Python. The code includes a convenience function to write some output since conn.execute() just returns an iterable object—it doesn't print anything.

Example 10-7. Simple SELECT statements in SQL

```
SELECT Name, Prototype FROM Enzyme LIMIT 5
SELECT * FROM Organism LIMIT 4 OFFSET 6
SELECT * FROM Organism
        ORDER BY Genus, Species, Subspecies
```

```
         LIMIT 4 OFFSET 6
SELECT COUNT(*) FROM Organism
```

Example 10-8. Simple SELECT statements using the sqlite3 module

```
dbname = 'path to database'
conn = sqlite3.connect(dbname)

def execute(query, args=[]):
    print()
    print(query)
    print()
    try:
        for result in conn.execute(query, args):
            print(result)
    except sqlite3.Error as ex:
        print(ex, file=sys.stderr)
        raise

for query in (
    'SELECT Name, Prototype FROM Enzyme LIMIT 5',
    'SELECT * FROM Organism LIMIT 4 OFFSET 6',
    '''SELECT * FROM Organism
           ORDER BY Genus, Species, Subspecies
           LIMIT 4 OFFSET 6''',
    'SELECT COUNT(*) FROM Organism',
    'SELECT MAX(Name, FROM Enzyme',
    'SELECT COUNT(Subspecies) FROM Organism',
    'SELECT COUNT(DISTINCT Genus) FROM Organism',
    '''SELECT DISTINCT Species FROM Organism
              WHERE Genus = 'Mycobacterium'
              ORDER BY Species, Subspecies ''',
    ):
    execute(query)

conn.close()
```

The results of executing the preceding script are shown next. The third query reorders the results first by genus, then species, and finally subspecies:

```
SELECT Name, Prototype FROM Enzyme LIMIT 5

('HpyF30I', 'TaqI')
('TmiI', '')
('BstD102I', 'BsrBI')
('BssHII', 'BsePI')
('BstXII', 'MboI')

SELECT * FROM Organism LIMIT 4 OFFSET 6

(7, 'Bacillus', 'stearothermophilus', '1473')
(8, 'Actinobacillus', 'suis', 'NH')
(9, 'Helicobacter', 'pylori', 'RFL21')
(10, 'Thermus', 'species', None)
```

That the results of the second query are ordered according to the numerical IDs we generated for them is an implementation detail that you can't generally rely on.

```
SELECT * FROM Organism
           ORDER BY Genus, Species, Subspecies
           LIMIT 4 OFFSET 6

(2400, 'Acetobacter', 'pasteurianus', None)
(1259, 'Acetobacter', 'pasteurianus', 'B')
(1538, 'Acetobacter', 'pasteurianus', 'C')
(1899, 'Acetobacter', 'pasteurianus', 'D')

SELECT COUNT(*) FROM Organism

(3031,)   # note that even single values are returned as tuples

SELECT MAX(Name) FROM Enzyme

('Zsp2I',)

SELECT COUNT(Subspecies) FROM Organism
        # the number of organisms whose subspecies is not NULL
(2530,)

SELECT COUNT(DISTINCT Genus) FROM Organism

(241,)
```

There are ordering dependencies among the kinds of clauses that can appear in a `sqlite3 SELECT` statement.[‖] Essentially, whatever clauses are contained in query must appear in the following order:

```
DISTINCT
EXPRESSION(s)
FROM
WHERE expression
GROUP BY column_names
ORDER BY column_names
LIMIT integer [OFFSET integer]
```

The definition of `execute` in Example 10-8 takes an optional second argument. This is for the values to be substituted into a parameterized query. Here are two of the queries we just showed, but in parameterized form. We use tuples to supply the query and the

‖ A diagram documenting all the variations on `SELECT` together with explanatory text is available at *http://www.sqlite.org/lang_select.html*. Because this is complete documentation much of it is more advanced than you will want to deal with, but you should be able to pick out the important parts.

values to be substituted. The first query uses a question mark as a placeholder, while the second uses the dictionary notation:

```
for query in (
    ("SELECT Name, Prototype FROM Enzyme LIMIT ?", (5,)),
    ("SELECT * FROM Organism LIMIT :lim OFFSET :off", {off: 6, lim: 5})
    ):
    execute(query[0], query[1])
```

Qualified database queries

Despite the variety of queries we've seen so far, the most significant part of a `SELECT` statement is the ability to restrict results based on column values. This is done using `WHERE` clauses and expressions. `WHERE` clauses follow the `FROM` clause(s) in `SELECT` statements.

SQL

WHERE Clauses in SELECT Statements

A basic `WHERE` clause has the form:

```
SELECT * FROM table_name WHERE expression
```

Often, the expression has multiple subexpressions connected by combinations of `AND` and `OR`:

```
SELECT * FROM tablename WHERE expression1 AND expression2
```

Example 10-9 shows two queries with `WHERE` clauses.

Example 10-9. WHERE clauses in SELECT statements

```
for query in (
    '''SELECT COUNT(*) FROM Enzyme WHERE Prototype IS NULL''',

    '''SELECT DISTINCT Species FROM Organism
             WHERE Genus = 'Mycobacterium'
             ORDER BY Species''',
    ):
    execute(query)
```

Here are the results:

```
SELECT COUNT(*) FROM Enzyme WHERE Prototype IS NULL

(606,)

SELECT DISTINCT Species FROM Organism
              WHERE Genus = 'Mycobacterium'
              ORDER BY Species

('avium',)
('butyricum',)
```

```
('chelonei',)
('fortuitum',)
('gordonae',)
('habana',)
```

Relationship queries

Consider these example queries performed in the `sqlite3` interactive command shell:

```
SELECT * FROM Organism WHERE OrgID BETWEEN 11 AND 14;
OrgID              Genus              Species            Subspecies
11                 Citrobacter        freundii           RFL22
12                 Streptomyces       luteoreticuli
13                 Streptomyces       albohelvatus
14                 Mycobacterium      butyricum

SELECT Name, OrgID FROM Enzyme WHERE OrgID BETWEEN 11 AND 14;
Name            OrgID
Cfr22I          11
SluI            12
SabI            13
M.MbuII         14
M.MbuIV         14
M.MbuI          14
M.MbuIII        14
```

The results of these queries show that rows in the Enzyme table with an OrgID value of `14` represent the enzymes produced by the organism whose OrgID is `14`. Suppose we wanted to start with the genus `Mycobacterium`, species `butyricum`, and subspecies `NULL` and find the enzymes that organism produces. First we can find the `OrgID` of the row representing the organism:

```
SELECT * FROM Organism
         WHERE Genus = Mycobacterium AND
         Species = butyricum AND
         Subspecies IS NULL;
OrgID              Genus              Species            Subspecies
14                 Mycobacterium      butyricum
```

Now that we have the organism's ID, we can use it in a query to get the enzymes it produces:

```
SELECT Name FROM Organism WHERE OrgID = 14
Name
M.MbuII
M.MbuIV
M.MbuI
M.MbuIII
```

We can write a function that generalizes these steps to return a list of the enzymes any organism produces, but it must handle an awkward problem. When using question marks in queries, the second argument to `execute` must be a sequence with as many elements as there are question marks in the query. However, to match a row where a column value is `NULL` requires the `WHERE` clause to include `IS NULL` for the column, rather

than `= ?`. Therefore, the function must prepare two different queries according to whether or not a value is supplied for the subspecies:#

```
def get_organism_enzymes(genus, species, subspecies=None):
    query = ('''SELECT OrgID from Organism
                WHERE genus = ? AND species = ? AND subspecies ''' +
             (' = ?' if subspecies else 'IS NULL'))
    rows = conn.execute(query,
                        (genus, species, subspecies) if subspecies
                        else (genus, species))
    if rows:
        lst = list(rows)                            # a tuple of tuples
        assert len(lst) == 1, len(lst)
        orgid_tuple = lst[0]                        # the tuple (orgid, ) retrieved by the
                                                    # first query
        assert len(orgid_tuple) == 1, len(orgid_tuple)
    return [row[0] for row in
            conn.execute('''SELECT name FROM Enzyme WHERE OrgID = ?''',
                         orgid_tuple)]
```

The first query gets the OrgID for the organism with the specified genus, species, and subspecies. The second query uses that ID to retrieve from the Enzyme table all enzymes with an OrgID that matches the one the first query retrieved. This corresponds to the interactive queries shown previously.

These examples demonstrate how 1-N relationships work. A primary key in one table is a foreign key in another table. While primary keys must be unique among the rows of a table, foreign keys are not restricted in that way. The rows with a particular value for their foreign key are the N rows of the 1-N relationship, with that value selecting a unique row of the other table. (There could be several foreign keys in one table; we're talking about just one here for simplicity's sake.) Figure 10-2 illustrates the process.

This kind of two-step lookup for 1-N relationships is so common that SQL provides syntax expressing it in a single query. The definition of `get_organism_enzymes` shown earlier can be greatly simplified by using this syntax:

```
def get_organism_enzymes_query(genus, species, subspecies=None):
    query = ('''SELECT Enzyme.name from Organism, Enzyme
                WHERE genus = :genus AND
                      species = :species AND
                      subspecies ''' +
             (' = :subspecies' if subspecies else 'IS NULL') +
             '''AND Organism.OrgID = Enzyme.OrgID''')
    return [row[0] for row in
            conn.execute(query,
                         {'genus': genus,
                          'species': species,
                          'subspecies': subspecies}
                         )]
```

#It isn't the question marks that are the problem—even if the function were to construct a literal query with no question marks, either by string concatenation or the string formatting mechanism, it would still have to construct a different query for when the subspecies is `None` and for when it's not.

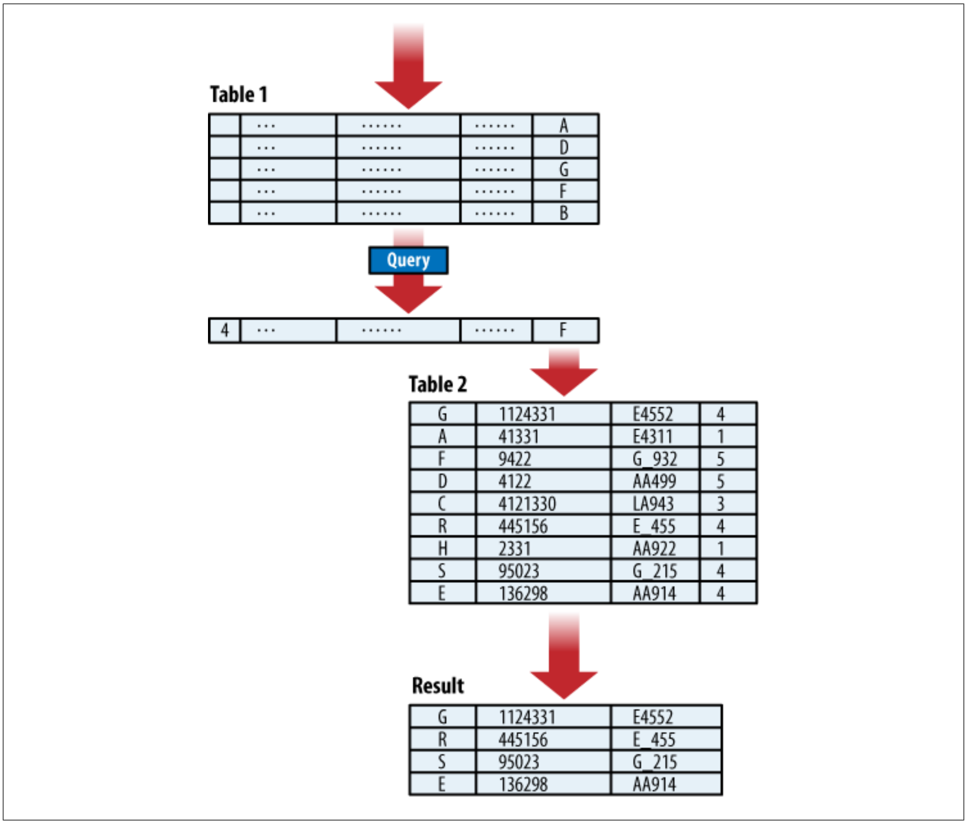


Figure 10-2. Navigating a 1-N relationship

This is the first query we've shown that accesses more than one table. Two tables may have a column with the same name, so to disambiguate which table's column is meant the name of the table and a period are added to the column name. (This is analogous to how names defined in a Python module are accessed by prefixing them with the module's name and a period.)

> Even if a column name is unique among the tables of the query, qualifying each column name with the name of its table can help make the query clearer.

The critical piece of the new query is the first line that's highlighted. It restricts the results to just those rows of the Enzyme table whose foreign key OrgID matches the OrgID of the row of the Organism table with the values of genus, species, and subspecies.

TEMPLATE

Querying 1-N Relationships

Given the following:

- *key* is the primary key of *table1*.
- *foreignkey* is the foreign key from *table2* to *table1*.
- *criteria1* is a part of a SELECT statement that obtains a single row from *table1*.
- *resultcolumns2* are the columns of *table2* to return as results.

queries to obtain those results take the following form:

```
SELECT resultcolumns2
       FROM table1, table2
       WHERE criteria1 AND
             tablename1.key = tablename2.foreignkey
```

The syntax shows only single-column keys, but multicolumn keys can be used by adding more AND phrases to the WHERE clause.

Working with M-N relationship tables is actually no different than working with the foreign keys from one table to another that implement 1-N relationships. In most respects an M-N relationship table just implements two separate 1-N relationships, one with each of the two tables it connects. Each of the two 1-N relationships uses a different foreign key. M-N relationships are really a design concept; the database mechanisms used to implement them are the same.

One difference does show up in queries involving M-N tables, though. We saw earlier how getting the "N" side of a 1-N relationship involves first finding the "1" row according to specified criteria, then getting each row from the "N" side whose foreign key corresponds to the primary key of the row on the "1" side. If we want to go from a row in one table to all the rows in another table by way of an M-N table, we have three tables to consider. Figure 10-3 illustrates the process.

Our Rebase database contains an M-N table that indicates what references apply to which enzymes. This must be an M-N table because an enzyme can have multiple references, and one reference can be for multiple enzymes. A typical query, though, would ask either for the references corresponding to a specific enzyme or for the enzymes corresponding to a specific reference:

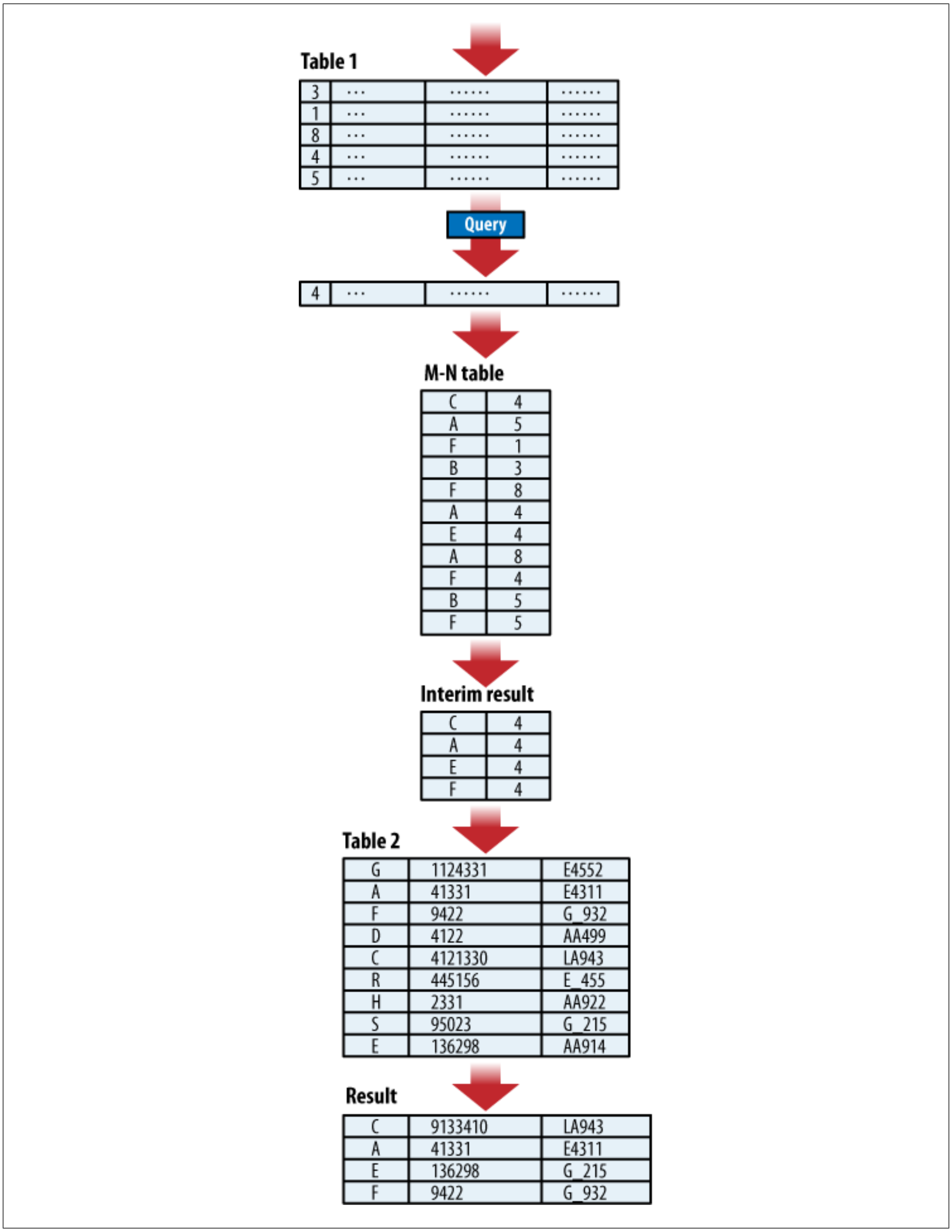


Figure 10-3. Navigating an M-N relationship

```
def get_references_for_enzyme(enzyme):
    """Return the reference details for the enzyme named enzyme"""
    return [row[0] for row in
            conn.execute('''
                SELECT Reference.details
                FROM Enzyme, Reference, EnzymeReference
                WHERE Enzyme.name = ? AND
                      Enzyme.name = EnzymeReference.Enzyme AND
                      Reference.refID = EnzymeReference.RefID
                      ''', (enzyme,))]

def get_enzymes_for_reference(ref):
    """Return the name of the enzymes referenced
    by the reference whose ID is ref"""
    return [row[0] for row in
            conn.execute('''
                SELECT Enzyme.name
                FROM Enzyme, Reference, EnzymeReference
                WHERE Reference.RefID = ? AND
                      Enzyme.name = EnzymeReference.Enzyme AND
                      Reference.refID = EnzymeReference.RefID
                      ''', (ref,))]
```

In either case, the "specific" item is obtained through the usual selection criteria, but its key column(s) match multiple rows of the M-N table rather than of another table from a 1-N relationship. Then, the foreign keys to the other side of the relationship are used to access rows in the other table.

The following template sums up the details of queries for M-N relationships.

TEMPLATE

Querying M-N Relationships

Given the following:

- *`key1`* is the primary key of *`table1`*.
- *`key2`* is the primary key of *`table2`*.
- *`table3`* represents an M-N relationship between *`table1`* and *`table2`*.
- *`foreignkey1`* is the foreign key from *`table3`* to *`table1`*.
- *`foreignkey2`* is the foreign key from *`table3`* to *`table2`*.
- *`criteria1`* is a part of a `SELECT` statement that obtains a single row from *`table1`*.
- *`resultcolumns2`* are the columns of *`table2`* to return as results.

the form of the query to get those results is as follows:

```
SELECT resultcolumns2
      FROM tablename1, tablename2, tablename3
      WHERE criteria1 AND
```

```
tablename1.key1 = tablename3.foreignkey1 AND
tablename2.key2 = tablename3.foreignkey2
```

The syntax shows only single-column keys, but multicolumn keys can be used by adding more `AND` phrases to the `WHERE` clause.

Querying the Database from a Web Page

In this final section of the chapter we'll put a web-based user interface in front of the database. There are a number of strategies for accomplishing this based on HTML forms. These include:

- Allowing the user to type an arbitrary query into a text box*
- JavaScript-driven dynamic pages that change what's presented in the form according to the information already entered, the way some widely used web search pages do
- Elaborate "query by example" dynamic web pages
- A fixed form that provides fields for specifying values for a particular query
- A form that presents a menu of query forms
- A page that presents multiple forms, each with fields for specifying values for a particular query

In order to keep the HTML details as simple as possible, we'll use the last approach.

HTML frames

We are going to use HTML *frames* to construct a page with a choice of queries on top and results on the bottom. Basic HTML frame arrangements are actually quite simple: the frame page just names the other pages and says how much horizontal or vertical space each should take. The amount of space is expressed either in pixels, as percentages of the parent page, or with an asterisk (meaning that it should take all the remaining space). Example 10-10 shows the HTML for our frame page.

Example 10-10. HTML page for Rebase browser-based queries

```
<html>
<head>
<title>Rebase Database Queries</title>
</head
  <frameset rows="400,*">
    <frame name='queries' src='allenz.html' scrolling='yes'>
    <frame name='results' scrolling='yes'>
```

* The obvious security problem of this approach can easily be obviated by requiring the first word of the query to be `SELECT`. The real problem with this approach is the requirement that the user know enough SQL to formulate the desired queries, as well as the schema of the target database.

```
  </frameset>
</html>
```

Note that this page does not include a body—a *frameset* is used instead. The `rows` attribute in the `<frameset>` tag specifies that the top frame should be 400 pixels high, and the bottom frame should occupy the rest of the window. Each of the two `<frame>` tags specifies a name and says to activate scrollbars when needed. The `src` attribute of the first `<frame>` tag gives the address (relative, in this case) of the web page to be displayed in that frame. The second frame starts out empty: it will be filled with the results of the queries performed in the first frame.

An advantage of this two-frame approach is that the contents of the top frame are stable, making it easier for a user to find queries within it. Another advantage is that the form does not have to be sent back to the browser with each query's results.† The results will replace the contents of the bottom frame, leaving the top frame unaffected.

An HTML page of query forms

The HTML page in the top frame will contain little more than a series of forms. We'll avoid discussing HTML tables and Cascading Style Sheets (CSS) and consider only the simplest possible construction (however, you can find several variations for this page that use tables, CSS, and even JavaScript in the code files accompanying the book). Each query will be displayed on the web page using HTML like that in Example 10-11.

Example 10-11. Simple HTML for query forms

```
❶ <b>Species for Genus</b>
❷   <form action='/cgi-bin/rebase' target='results'>
❸   <input type='hidden' name='query' value='species_for_genus'>
❹     genus <input name='genus'> (e.g. Vibrio)<br/>
❺     <p><input type='submit'></p>
❻   </form>
```

This will display in a browser something like what's shown in Figure 10-4. Here's a line-by-line description of what the HTML means:

❶ The title of the query, in bold.

❷ The beginning of a form (on a new line).

   a. The form's action is the program *rebase* found in the *cgi-bin* subdirectory of the directory from which the server is running.‡

† This was discussed at the end of the previous chapter; see Example 9-20 ("HTML forms with CGI scripts" on page 350).

‡ See "Serving CGI requests" on page 343 for instructions on how to run a simple CGI server using Python's `http.server` module.

b. The form specifies the target `'results'`, which is the name of the other frame; if no target were specified the results of the query would replace the page with the form in the browser.

❸ An input field with type `'hidden'`; this kind of field is not displayed but *is* included with the rest of the fields when the form is submitted—in effect, it is an invisible constant field. It is this field that tells the Python program which SQL query to execute.

❹ A label, a one-line text input field, and an example to show the user the sort of value the query expects the field to contain.

❺ The submit button; when this is pressed, the form's action is submitted to the web server along with the field values. With no `'value'` attribute the button will have a default label, which for most browsers will be "Submit."

❻ The end of the form.

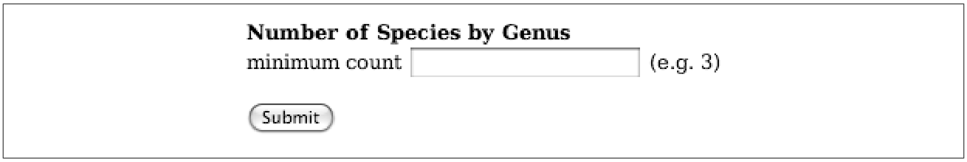


Figure 10-4. Display of a simple query form

Multiple forms can appear on the same page, as Example 10-12 illustrates. Pressing a submit button invokes the action of the form that contains it with the values of that form's fields. There is no connection to the fields of other forms on the same page.

Example 10-12. A page of Rebase queries

```
<html>
<head>
<title>Rebase Queries</title>
</head>
<body>
<b>Number of Species by Genus</b>
  <form action='cgi-bin/rebase' target='results'>
  <input type='hidden' name='query' value='number_of_species_by_genus'>
    minimum count <input name='minimum_count'> (e.g. 3)<br/>
    <p><input type='submit'></p>
  </form>
<b>Number of Enzymes by Organism</b>
  <form action='cgi-bin/rebase' target='results'>
  <input type='hidden' name='query' value='number_of_enzymes_by_organism'>
    minimum count <input name='minimum_count'> (e.g. 3)<br/>
    <p><input type='submit'></p>
  </form>
<b>Species for Genus</b>
  <form action='cgi-bin/rebase' target='results'>
  <input type='hidden' name='query' value='species_for_genus'>
```

```
    genus <input name='genus'> (e.g. Vibrio)<br/>
    <p><input type='submit'></p>
  </form>
<b>Subpecies for Genus and Species</b>
  <form action='cgi-bin/rebase' target='results'>
  <input type='hidden' name='query' value='subspecies_for_genus_species'>
    genus <input name='genus'> (e.g. Thermus)<br/>
    species <input name='species'> (e.g. thermophilus)
    <p><input type='submit'></p>
  </form>
<b>Enzymes for Organism</b>
  <form action='cgi-bin/rebase' target='results'>
  <input type='hidden' name='query' value='enzymes_for_organism'>
    genus <input name='genus'> (e.g. Helicobacter)<br/>
    species <input name='species'> (e.g. pylori)<br/>
    (optional) subspecies <input name='subspecies'> (e.g. RFL44)
    <p><input type='submit'></p>
  </form>
<b>Prototype Enzymes</b>
  <form action='cgi-bin/rebase' target='results'>
    <input type='hidden' name='query' value='prototype_enzymes_like'>
    name like <input name='name_like'> (e.g. Ha%)<br/>
    <p><input type='submit'></p>
  </form>
<b>References for Enzyme</b>
  <form action='cgi-bin/rebase' target='results'>
  <input type='hidden' name='query' value='references_for_enzyme'>
    enzyme <input name='enzyme'> (e.g. EcoRI)<br/>
    <p><input type='submit'></p>
  </form>
<b>Enzymes for Reference Number</b>
  <form action='cgi-bin/rebase' target='results'>
  <input type='hidden' name='query' value='enzymes_for_reference'>
    reference number <input name='number'> (e.g. 11)
    <p><input type='submit'></p>
  </form>
</body>
</html>
```

This page by itself will display as in Figure 10-5. The frame-based web page is shown in Figure 10-6.

Tips, Traps, and Tracebacks

Tips

- Just as you should develop and debug regular expressions using a testing tool like **re-try**, it is often helpful to use a tool to explore a database directly. If your computer has the command-line command `sqlite3` installed, you could use that. (If it doesn't, you can try downloading the appropriate file for your platform from

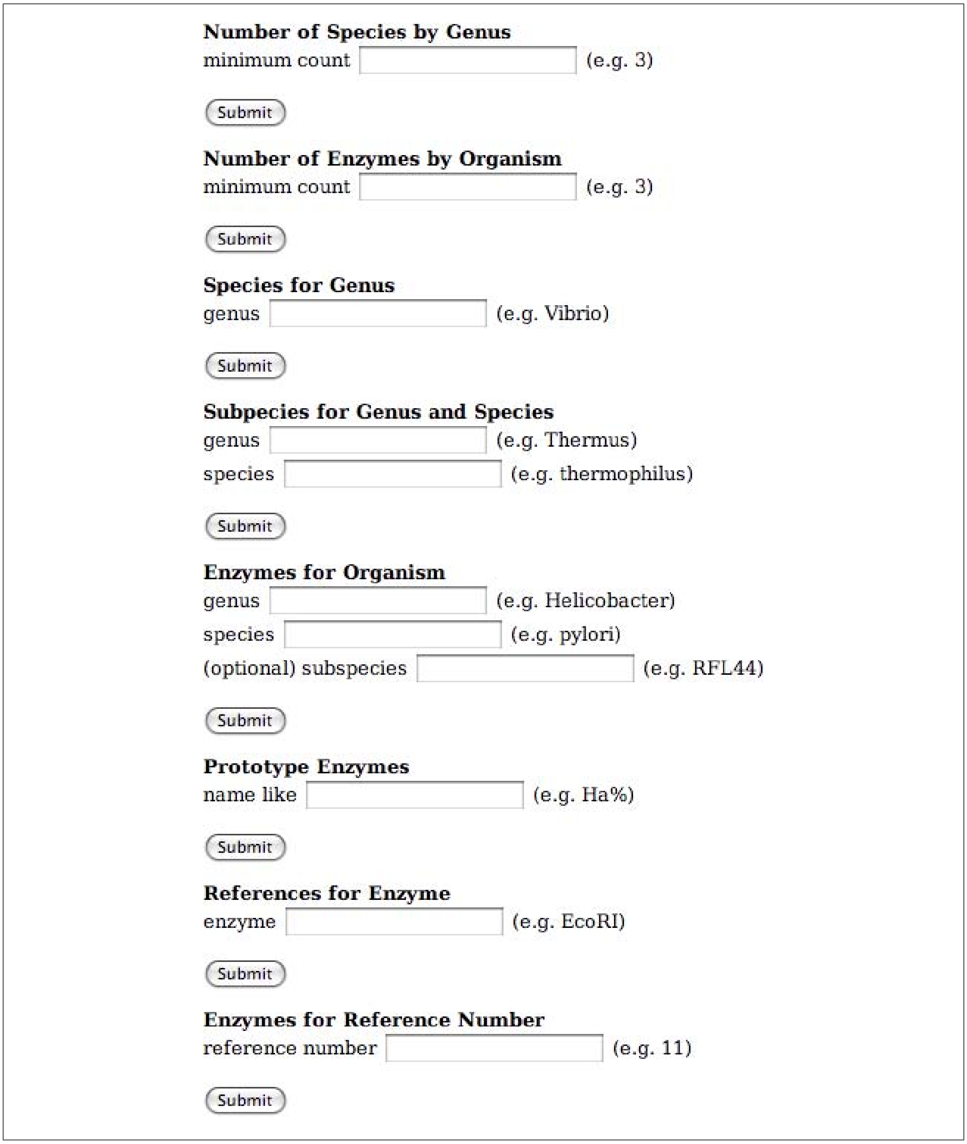


Figure 10-5. Display of a page of query forms

http://www.sqlite.org/download.html—you want just the file that contains the precompiled command-line installation, or, if there is none for your platform, the source code.) There are many GUI interfaces—some free, some commercial; some platform-independent, some platform-specific. You can find a long list of information about them at *http://www.sqlite.org/cvstrac/wiki?p=ManagementTools*. A particularly convenient, simple, free, and cross-platform GUI interface to

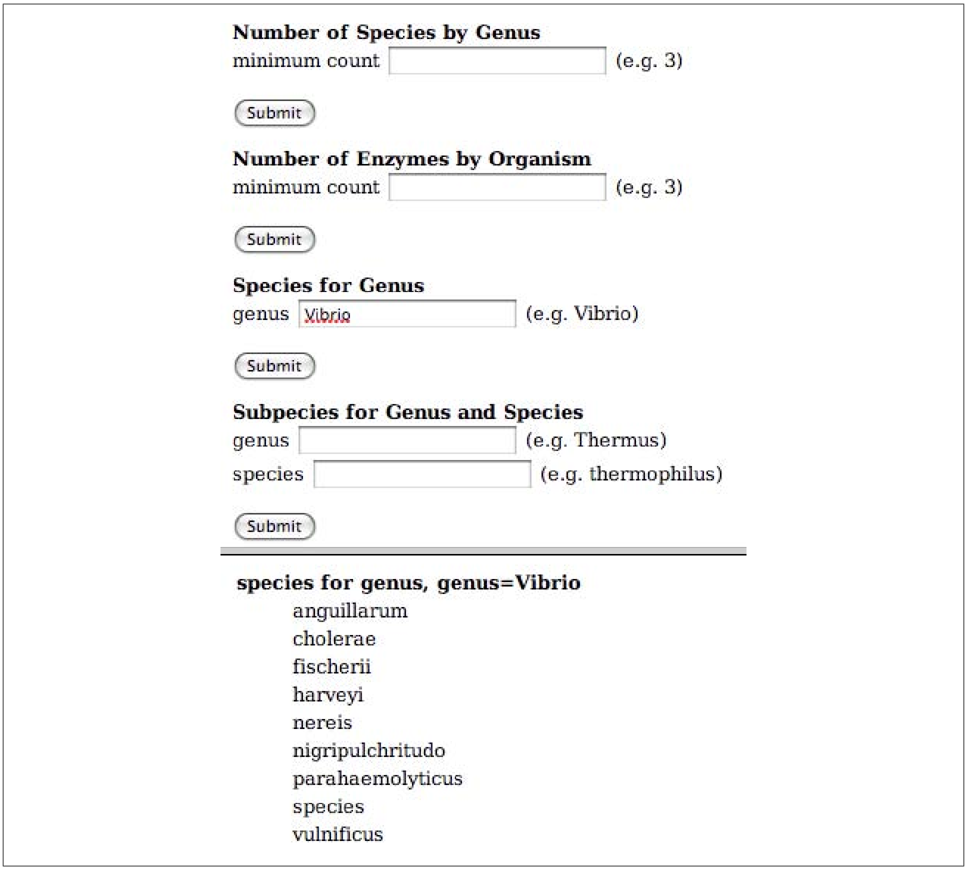


Figure 10-6. Frame-based display of a page of query forms

`sqlite3` is the SQLite Database Browser, which you can download from *http://sqlitebrowser.sourceforge.net/index.html*.

- You can save yourself a lot of trouble by always using single-column integer-valued primary keys in relational database tables. This chapter's examples used some other kinds in order to demonstrate how keys work, but in practice there is a major problem with using meaningful column values as keys: you may need to change those values while curating the database (e.g., to correct inconsistent abbreviations, spelling, or names), but database systems generally do not allow changes to the values in primary key columns. Even if the database system did allow changes to foreign key values, making them would require the same changes to be made in all the rows of any other tables that use the key values as foreign keys. Using integer-valued primary keys—preferably ones that are automatically generated by the database system—avoids these problems.

- The implementation of SQLite3 makes using an integer as a primary key easy, because every table has a key called `ROWID` that is automatically incremented every time a row is inserted into the table.
- You can simplify calls to `Connection.execute` by naming query strings elsewhere and just using their names when calling `execute`. This also has the advantage that the same query string can be used in multiple places.

Traps

- `sqlite3.Connection.close` does *not* do a commit; you must call `commit` before closing the connection if you want the changes to be committed.
- If you are working in the interpreter and don't close a `sqlite3` database connection, attempting to open it again will raise an error. This can happen very easily when you are debugging your code. You should always close connections in the `finally` of a `try` statement.

Tracebacks

Error messages you might encounter while using the `sqlite3` module include the following:

`sqlite3.Error`
: An error has been produced by the `sqlite3` module.

`sqlite3.OperationalError`
: An attempt has been made to execute an invalid SQL statement.

`SyntaxError: invalid syntax`
: A string passed to `eval` contained a syntax error. (Python must parse the string before it can execute it.)

CHAPTER 11
Structured Graphics

Bioinformatics deals with enormous amounts of data. People process visual imagery far more easily and quickly than words and numbers, so many bioinformatics websites and applications provide highly effective visualizations of complex data. Generating graphical presentations is an important aspect of bioinformatics programming. This chapter will show you how to do it.

We will look at implementations of three kinds of information displays:

- *Histograms* are useful for anything involving counts. They are used widely in many fields and business operations. In bioinformatics, they are frequently used to provide high-level visual overviews of quantitative distributions.
- *Dot plots* are also a general-purpose chart type, but in bioinformatics they have special uses for visualizing how two sequences—or a sequence and itself—are related.
- *Raw data* collected from laboratory devices such as sequencing machines is usually displayed as curves. The classic example is the four-colored "trace" of the data from a traditional sequencing device.

The data used for all of the examples can be found on the book's website. In writing your own applications, you can obtain similar data via URL queries.

Introduction to Graphics Programming

Computers represent visual images in one of two ways:

Bitmaps
: A bitmap is a two-dimensional arrangement of *pixels*, each representing the color of a "dot" at a particular location in the image.

Structured graphics
: A structured image is a set of instructions in a "drawing language" that describe individual objects representing different kinds of shapes and text.

You may already be familiar with this distinction from having used some applications that manipulate bitmaps and others that manipulate diagrams. Photographs, icons, screen snapshots, and the contents of the display itself are represented as bitmaps. Images in which you can select individual components, such as a rectangle or piece of text, are structured. You can add, remove, and group components in a structured image, and change a given component's properties (color, location, border width, etc.).

A structured image may be converted to a bitmap, as when a snapshot is taken of a window on the screen displaying a diagram. Bitmaps, though, cannot be turned into structured representations, because they do not contain any information about the individual components of the image.

Concepts

It is difficult to talk about graphics programming in the abstract. Structured graphics are implemented in a variety of ways, using numerous file formats, languages, libraries, and tools. Some technologies store images using file formats resembling the object persistence mechanisms discussed at the end of Chapter 6, while others represent them in an XML-based format. Still others are designed to display images in windows on the screen.

For the most part, all graphics programming technologies share a common set of concepts and mechanisms. However, the vocabulary and technical details used in working with these concepts and mechanisms can vary significantly from one technology to another. Learning to program in your first graphics technology can be quite disconcerting because you must learn not only a new conceptual framework and new terminology, but also all the details of that particular technology.

Structured graphics can be discussed and even programmed without anything more than a vague idea of what output device will be used to display them. In fact, a diagram or image can be generated without any display at all. The result is either a file written in some structured graphics format, a textual description written in a well-defined graphics language, or simply a data structure inside the computer's memory. Outputting the graphics to a screen, printer, or other device is essentially an independent step. The three main aspects of all structured graphics tools are their *coordinate system*, *components*, and component *properties*.

Coordinate system

Structured graphics are created within a coordinate system—a grid of points referenced by <x, y> pairs. In most graphics technologies, the upper-left corner of the window is at (0,0) and the coordinates are positive numbers (not necessarily integers). This corner is typically the most stable, given the way window systems and interfaces handle changes to window size: dragging the lower-right corner with the mouse is probably the most common resize action across all operating systems, and that leaves the upper-left corner unaffected. Even where a system or application provides a way to resize a

window at the top or left, the contents usually stay at the resulting upper-left corner. This is illustrated in Figure 11-1.

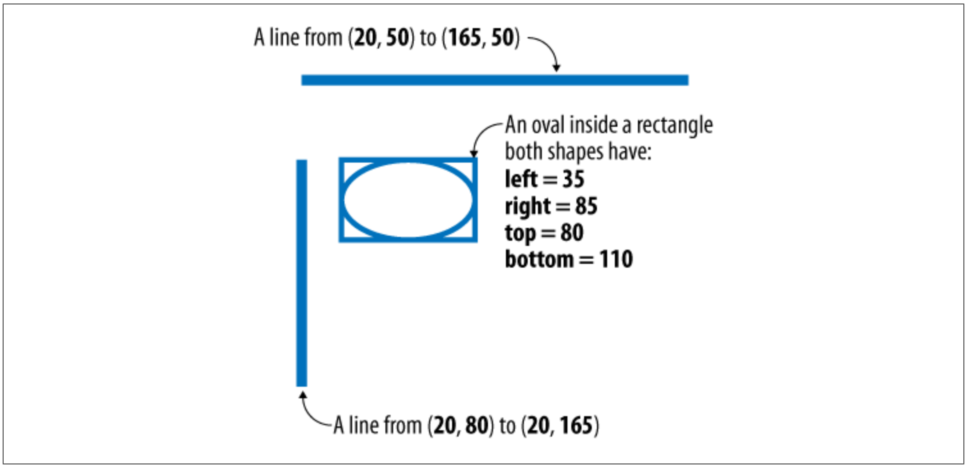


Figure 11-1. Illustration of coordinate system

Unfortunately for beginners, this coordinate system does not correspond to any of the quadrants of the traditional Cartesian system. The position of the origin relative to the quadrant corresponds to the lower-right Cartesian quadrant, in which x is positive and y negative. For designing graphical user interfaces, this can be somewhat disorienting at first. And for graphics representing quantitative data, it's *really* confusing, because every y coordinate must be negated. That is, the heights of titles, legends, and other parts of a diagram that are not part of the quantitative display must be *subtracted* from the y coordinates of each data point.

The basic unit of a coordinate system is the *pixel*. For most purposes, "pixel" means the smallest dot a device can display independently. Computer display measurements are an example: 1280×1024 means that the computer screen has 1,280 columns and 1024 rows of individually addressable points. However, all modern technologies that involve "layout"—graphics, word processing, web pages, etc.—allow measures to be specified in different units. Table 11-1 lists the measurements implemented by most systems, along with their common abbreviations. Of course, not every system supports every measure or every abbreviation.

Table 11-1. Units of measurement used in software

Name	Abbreviations	Meaning of 1 of the unit
Pixel (default)	`px`	Smallest unit of the display device
Inch	`i, in`	Number of "dots per inch" of the display device
Millimeter	`mm`	Relative to inch

Name	Abbreviations	Meaning of 1 of the unit
Centimeter	`cm`	Relative to inch
Point[a]	`p`, `pt`	1/72 inch
Pica	`P`, `pc`	12 points, 1/6 inch
Em	`em`	Point size of the component's font
Ex	`ex`	x-height of the current font

[a] "Point" and "pica" are printers' terms with a very long history and slightly different values across countries and industries. Modern computer technologies have standardized on the definition of a "point" as 1/72 inch.

A component's *bounding box* is the rectangle that minimally encloses the component within the coordinate system. For basic shapes it is obvious what the bounding box is, but this is not always the case for more complicated shapes. Depending on the technology, the edge of the bounding box may include one or more spaces designated as the *border*, *padding*, and *margin*. The "border" is a line drawn on a bounding box edge. The width of the line and its color can be specified. "Padding" usually adds extra space between what the component requires and its actual bounding box. The "margin" usually refers to extra space between a component's bounding box and a neighboring component.

Some graphics systems allow the width of each of these spaces to be specified separately for each of the four sides of the bounding box, while others allow only horizontal and vertical distances to be specified. Some systems allow a color to be specified for margins and/or padding as well as for borders. The definitions of the "height" and "width" of a component, while related to its bounding box, also differ from system to system depending on whether they include padding, borders, and/or margins.

Components

The primitive components of structured images are text items and a variety of shapes. For each, there is at least one point that specifies its location. Other points and lengths designate other aspects of its geometry. The components are defined as follows:

Text
: An individual piece of text usually "wrapped" to fit within its bounding box's width

Line
: From one point to another

Polyline
: A series of points, with a line drawn from each to the next; in effect, a composite of line segments where each one's end point is the same as the next one's start point

Rectangle
: Expressed using either two opposite corners or as a corner and a width and height

Polygon
: A closed shape defined by a series of points, with a line drawn from each point to the next and from the last to the first

Circle
: Defined using a center point and radius length

Oval
: May be defined by its rectangular bounding box or by its center point and its vertical and horizontal radii

Arc
: A segment of an oval specified in one of a number of ways (e.g., specifying the oval along with a start and stop angle in degrees); whether the arc is meant to correspond to the smaller portion of the oval or the larger must also be indicated

Cubic spline
: A curve defined by a start point, an end point, and one control point

Quadratic spline
: A curve defined by a start point, an end point, and two control points

Composite
: Several components grouped to form a new component

Composition is recursive: since composites are components, they can be included in other composites. A "top-level" component is one that is not included in a composite. Top-level components are not necessarily composites, though. For instance, a drawing may consist of a rectangle and a composite containing two lines; such a drawing would consist of three shapes and two top-level components: a rectangle and a composite.

Properties

Text and most shapes have three areas that can be controlled separately with respect to color: *foreground*, *background*, and *border*. Depending on the shape and the area, a "pattern"—a bitmap or image displayed repeatedly—may be used in place of a color. The color property may be called *fill* or *stroke* depending on the technology and whether or not the shape has area (a rectangle versus a line, for example).

Text objects also have fonts and other properties regular shapes do not. Some, but not all, technologies allow changes to font properties within a text object. If they do not, a composite can be used to combine two text objects with different font properties.

Other properties are specific to one or more shapes. For example, a rectangle may have a rounding angle to specify how its corners are drawn, and an arc defined as a segment of an oval will have properties such as an angle to specify what part of the ellipse it includes and a starting point along the ellipse.

Properties are generally optional when creating a component, and most properties can be modified once a component has been created (either to refine the layout while the diagram is being constructed or to add some dynamic aspects to the display).

GUI Toolkits

Window-based technologies do more than just support structured graphics. They are complete toolkits for building graphical user interfaces (GUIs).

The primary purposes of a graphical user interface are to display content—documents, images, etc.—to the user and to provide mechanisms—menus, buttons, scrollbars, input fields, etc.—for invoking application actions. The term *widget* is used as a general noncommittal term to refer to controls and other interface components.

Very few applications provide GUIs for structured graphics content. However, toolkits generally provide a *canvas* widget that supports structured drawing. Canvases can be added and displayed along with the rest of the interface and content.

GUI toolkits are not usually the best way to generate images for visualizing bioinformatics data. They often do more than is needed, and they may be difficult to learn to use. Other reasons why a GUI toolkit might not be appropriate include:

- The application may produce image files without displaying anything, often executing with no user interaction.
- The application may generate images to be displayed in a web browser, in which case the browser supplies the GUI technology.
- The application may simply need to display an image in a window, in which case you don't need buttons, menus, and all the other elaborate features of a GUI—your operating system's mechanisms for managing windows are good enough.

Still, it can be useful to learn the basic details of how to display an image on the screen using a particular toolkit. Just because it provides elaborate controls and fancy features doesn't mean you have to use them.

GUI toolkits for Python

Python does not implement its own GUI toolkit. Instead, you will use one of four popular platform-independent graphical user interface libraries, along with a Python interface. This is analogous to how SQLite3 and other relational databases are used from Python. As with the `sqlite3` module, the interface for one of these—the `tkinter` module interface to Tk—is built into Python.

The four toolkits, along with the URLs to their home pages and the corresponding Python interface libraries, are listed in Table 11-2.

Table 11-2. Cross-platform GUI toolkits and their Python interfaces

Toolkit	Home page	Python toolkit	Python interface home page
Tk	*http://activestate.com/activetcl*	`tkinter`	`tkinter` module documentation
wxWidgets	*http://wxwidgets.org*	wxPython	*http://wxpython.org*
Qt	*http://qt.nokia.com*	PyQt	*http://riverbankcomputing.com/software/pyqt/intro*
GTK+	*http://gtk.org*	PyGTK	*http://pygtk.org*

One or more of these toolkits may already be installed on your computer. If not, you can get installers and source code from the toolkits' home pages. As of November 2009, both `tkinter` and PyQt have been updated for Python 3, but wxPython and PyGTK have not. Tk itself is not part of `tkinter`, so if it's not on your system you'll need to go to the web page shown in the table to download an installer. Even if it is on your system, it may be an older version; if you get error messages about version incompatibilities, download and install the latest version.

A great deal of material has been published in books and on the Web about each of these cross-platform toolkits. You should be able to find tutorials, examples, forums, and many other sorts of resources that can help you get started with any of these and solve any problems you may encounter when using them later.

If you want to experiment with building applications with user interfaces, start with the `tkinter` documentation and the resources listed at its beginning. IDLE is implemented using `tkinter`. You can find its code in a subdirectory called *idlelib* in your Python installation's library directory. That will give you a large body of `tkinter` code to read.

Using a GUI toolkit

Interfaces using GUI toolkits have several high-level layers. As with structured graphics components, the details vary among toolkits. In general, there is an outermost structure called a *frame* that contains everything displayed—both controls and content. Around the frame is the window, managed by the operating system. Inside the frame are one or more *geometry managers*, or, as they are sometimes called, *layout managers*.

Every control and content item, including canvases, is added to a parent item, with the frame acting as the top-level parent. The layout manager automates the work of arranging their exact placement according to its *policy*. Policies are provided for putting each item next to the previous one either horizontally or vertically, arranging the items in a grid, and so on. Some toolkits provide a wide range of layout managers and policies, and others provide only a few. Some require layout managers to be explicitly created and added to frames, while others automatically create a layout manager for each frame. Many toolkits require a call to a function for the layout manager to do its work once its elements have all been added. Others require a call to a function for each element added.

To connect the GUI to the application, the application needs to initialize the toolkit, add the necessary elements (frames, layout managers, controls, and content), then hand control to the toolkit. As a result, GUI-based applications are largely event-driven. The toolkit waits for the user to press a key, move the cursor, or click a mouse or trackpad button. When it receives such an "event," it invokes an application action (the application establishes which action will be invoked when each control is created). The wait-event-act cycle is usually called the *event loop*, for obvious reasons, and the toolkit will include a high-level function that the application calls to start the cycle.

Using a GUI toolkit simply to display images on the screen typically requires only a few steps (and not all of these will be necessary in every toolkit):

1. Initialize the toolkit.
2. Create a frame.
3. Create a layout manager for the frame.
4. Add a canvas to the layout manager.
5. Create shapes and text on the canvas.
6. Invoke the event loop function.

Structured Graphics with tkinter

The introduction to `tkinter` found here is far from complete. Remember that `tkinter` is meant to be a full GUI toolkit, not just one for constructing graphics. The only one of its "widgets" we really need for our current purposes is its `Canvas` class. We will ignore controls entirely: our programs will simply put up a display and take it down when the user hits the Return key.

tkinter Fundamentals

There are several ways to organize a `tkinter`-based program. You can just use functions, or you can define a class. If you define a class, it can be independent of `tkinter` or it can inherit from `Frame`. Inheriting from `Frame` would add very little to the way we will be using `tkinter`, but classes are very useful in graphics programming, so we'll use independent classes.

The basic steps

The high-level steps of a minimal `tkinter` structured graphics program are:

1. `root = tkinter.Tk()`

   Initialize `tkinter` and get a top-level instance.
2. `root.title(`*`string`*`)`

   Give the window the title *`string`*.

3. `canvas = tkinter.Canvas(root)`

   Create the canvas under the *root*.*

4. `canvas.pack()`

   Call the Packer geometry manager to display the canvas.

5. Call `Canvas` drawing methods on the canvas.

6. `root.close()`

   Close the window.

An instance of `Tk` represents a Tk environment. An application may have multiple instances of `Tk` open at the same time. As soon as a `Tk` instance is created, a small window appears on the screen.

There is a class for each kind of `tkinter` widget. The examples that follow use just `Canvas`. The first initialization parameter of each `tkinter` widget class is the existing container to which the new object should be added. This is called its "master" and must be the top-level `Tk` object or one of the very few other "container" components `tkinter` provides. The examples put the canvas directly into the top-level `Tk` instance.

Except for the master, all parameters in widget-creation calls are optional. These parameters specify details about the object being created. All have reasonable defaults.

Common widget options

Frequently used widget options are shown in Table 11-3. These options can be specified as keyword arguments when the widget is created, or they can be changed later with a call to the widget's `configure` method. Each of these options is supported by most, if not all, of the widgets.

Table 11-3. Frequently used tkinter widget options

Name	Keyword	Value type	Units (first is default)
Background color	`bg`	Color value	An X11 color name or a 6-hexadecimal-digit "RGB" string (color name) `'#RRGGBB'`
Foreground color	`fg`	Color value	
Border width	`bd`	Distance	(px), p(t), i(n), c(m), m(m)
Width	`width`	Integer	Relative to width (height) of average character or widget's font
Height	`height`	Integer	
Font	`font`	Font	String or tuple; see following discussion
Anchor	`anchor`	Direction	`'center'`,`'n'`,`'e'`,`'w'`,`'s'`,`'ne'`,`'nw'`,`'se'`,`'sw'`

* Either a dictionary or a series of keyword arguments may follow *root* to specify any of the following canvas properties: `background`, `bd`, `bg`, `borderwidth`, `closeenough`, `confine`, `cursor`, `height`, `highlightbackground`, `highlightcolor`, `highlightthickness`, `insertbackground`, and `insertborderwidth` (for example, `tkinter.Canvas(root, bg='gray', bd=4)`).

The `anchor` parameter specifies the alignment of a widget's content within the widget. Its value indicates a location relative to the content, which will be aligned with a corresponding location in the widget. For instance, `'nw'` means that the upper-left corner of the content will be aligned with the upper-left corner of the widget.

The `'#RRGGBB'` form of color specification works the way it does in HTML and CSS. (The three-digit form is not acceptable.) Each pair of hexadecimal digits represents an integer between `0` and `255`. White would be `'#FFFFFF'`, black `'#000000'`, and red `'#FF0000'`. X11 color names come from a large set that predates the Web.† For grayscale graphics it is worth noting that the names include `'gray0'` (black) through `'gray100'` (white), in increments of 1.

Fonts are specified either as a family name (e.g., `'Futura'`) or as a tuple of the form:

```
(family, size[, variation1[, variation2[, ...]]])
```

In the tuple, ***`family`*** can be either a single family name or a sequence of family names. If a sequence, `tkinter` will attempt to locate each one in the operating system environment in turn until a match is found. The variations are values such as `'bold'`, `'italic'`, and `'underline'`. Any number of variations may be present.

Canvas drawing methods

The `Canvas` class implements six drawing methods. You can use `pdb` to step through the drawing actions; as each item is drawn it will appear immediately in the canvas. These are not actually widgets—they are not part of `tkinter`'s widget class hierarchy, and they are accessible only through their canvas. Within the canvas, though, they behave very much like regular widgets in most respects. The methods are:

`create_text(`*`x`*`, `*`y`*`, `*`string`*`[, `*`keyword-arguments`*`])`
: Draw *`string`* at `(`*`x`*`,`*`y`*`)`.

`create_line(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`[[, . . .], `*`keyword-arguments`*`])`
: Draw a line from `(`*`x1`*`,`*`y1`*`)` to `(`*`x2`*`,`*`y2`*`)` and so on, for however many points there are (used for both lines and polylines).

`create_rectangle(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`[, `*`keyword-arguments`*`])`
: Draw a rectangle with `(`*`x1`*`,`*`y1`*`)` and `(`*`x2`*`,`*`y2`*`)` as opposite corners.

`create_polygon(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`, `*`x3`*`, `*`y3`*`[[, . . .], `*`keyword-arguments`*`])`
: Draw a polygon described by the line from `(`*`x1`*`,`*`y1`*`)` to `(`*`x2`*`,`*`y2`*`)` and then to each of the other points, finally drawing a line from the last point to the first.

`create_oval(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`[, `*`keyword-arguments`*`])`
: Create an oval inside the rectangle defined by opposite points `(`*`x1`*`,`*`y1`*`)` and `(`*`x2`*`,`*`y2`*`)`.

`create_arc(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`[, `*`keyword-arguments`*`])`
: Create an arc inside the rectangle defined by opposite points `(`*`x1`*`,`*`y1`*`)` and `(`*`x2`*`,`*`y2`*`)`.

† See *http://www.tcl.tk/man/tcl8.5/TkCmd/colors.htm* for a list of names and RGB values.

Table 11-4 shows some of the keyword arguments commonly used when creating and modifying objects on `tkinter` canvases. They are similar to those used for widgets in general, but some have different names and meanings.

Table 11-4. General canvas object properties

Name	Keyword	Value type
Inside color	`fill`	Color value
Border color	`outline`	Color value
Border width	`width`	Distance

Note that the width and height of a graphic object on a `Canvas` are not specified directly: they are derived from the x and y values in the call to the drawing method.

Each kind of drawing object also has its own specific options. Table 11-5 shows the ones we will use for text objects.

Table 11-5. Canvas text properties

Name	Keyword	Value type
Text	`text`	String
Font	`font`	Font
Anchor	`anchor`	Direction

Example 11-1 is a small program that demonstrates the actions to set up a `tkinter` window with just a canvas and an example of each kind of drawing operation. At the end, it asks the user to press the Return key, at which point it closes the window.

Example 11-1. Example tkinter program for structured graphics

```
colors = ('gray80', 'black', 'black', 'gray30',
          'gray50', 'gray70', 'gray30', 'black')
import tkinter
tk = tkinter.Tk()                   # initialize tkinter
tk.title('demo')                    # give window a title
canvas = tkinter.Canvas(tk)         # create the canvas
canvas.pack()                       # use the Packer geometry manager
canvas.create_rectangle(20, 10, 120, 80, fill=colors[0])
canvas.create_line(20, 100, 120, 100, width=4, fill=colors[1])
canvas.create_line(20, 170, 45, 115, 70, 170, 95, 115,
                   120, 170, fill=colors[2])
canvas.create_polygon(150, 180, 175, 115, 190, 160, 215, 125,
                       240, 180, fill=colors[3])
canvas.create_oval(180, 55, 250, 115, fill=colors[4])
canvas.create_arc(140, 10, 240, 80, extent=240,
                  width=4, fill=colors[5], outline=colors[6])
canvas.create_text(20, 190, text='Drawing on a tkinter canvas',
                   anchor='w', font=(('Verdana', 'sans'), 14, 'italic'),
                   fill=colors[7])
```

```
input('Press the Return key to close the window(s)')
tk.destroy()                                # close the window
```

Figure 11-2 shows the result of running this little program.

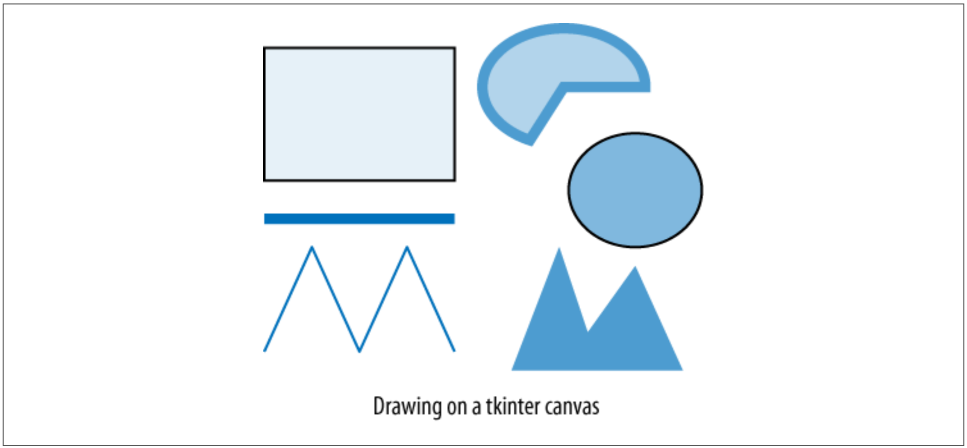


Figure 11-2. A small tkinter example

Writing the contents of a canvas to a PostScript file

I mentioned earlier that some structured graphics applications write image descriptions to files, and others write them to the screen. The `Canvas` class makes `tkinter` a bit of a hybrid, because it has the ability to write a PostScript description of its contents to a file. The PostScript file can then be sent to a printer, viewed on the screen with software that can display files in that format, or converted to another format by one of the many available image-conversion tools. A single method call suffices, specifying the filename, the width and height (in pixels) of the area to be captured, and whether the output should be in black and white (mono), gray, or color:

```
canvas.postscript(file=filename, width=width, height=height,
                  colormode='gray') # 'color', 'gray', or 'mono'
```

The background of the PostScript image will be white, ignoring the *`bg`* setting for the canvas. You may find when you view the resulting PostScript image, or a version of it converted into another format, that it is not the same size as the `tkinter` image was on the screen. You can scale the image proportionally in both horizontal and vertical dimensions by specifying either the `pagewidth` or the `pageheight` keyword argument, as either an `int` or a `float` (if you specify both, the value of `pagewidth` will be used and `pageheight` will be ignored). For instance, adding `pagewidth=0.6*width` to the preceding call would reduce the size of the PostScript image by 60% in each dimension. Note that `pagewidth` and `pageheight` represent the size of the image on the page—e.g., `800`—not a percentage: the example shows how to scale an image whose width you already know.

If the purpose of your graphics program is simply to produce the PostScript file and not to display anything on the screen, then add the following after the call to `canvas.postscript`:

```
tk.destroy()
```

This kills the frame and all of its content. In a reasonably short program, the canvas will pack, the PostScript file will be written, and the frame and canvas will be destroyed before the window even appears on the screen. For a longer program, the frame may appear briefly in a window, though not necessarily with its canvas contents.

Examples

We're going to look now at a class that uses `tkinter` to draw dot plots, and another that produces histograms. Despite their different content, the steps required to draw them and many of their details are quite similar. Even where the details vary, we will be able to parameterize them. Therefore, we will first develop an abstract class that will be used as the superclass of classes implemented to draw specific kinds of visualizations.

An abstract class for tkinter graphics

Building a full-featured class that could handle a wide variety of graphical content is a far larger undertaking than would be appropriate here. Besides, such a facility would involve a number of drawing and component classes, not just a single class.‡ The purpose of this exercise is simply to demonstrate the power of using inheritance to implement a framework (we saw an example of this with `HTML.parser`, in "Structured HTML parsing: html.parser" on page 298). It's also worth pointing out that one normally doesn't start by developing such a class. Rather, a particular example is developed, then another, and perhaps a third. Only as the similarities become apparent are they abstracted into a new shared superclass.

Before we look at the class itself, let's examine a short test script found at the end of its module. Example 11-2 contains the code. The class field and the two methods defined in its `EmptyPlot` class override those inherited from `Plot`. Note that since the class does not override `__init__`, the definition of `__init__` it inherits from its superclass gets called. The `execute` method called to display the plot is defined in the abstract class.

‡ There is an excellent library called *PyChart* that produces very high-quality graphics in Encapsulated PostScript (for publishing), PDF, PNG, and SVG. It uses Python only, so it doesn't require anything else to be installed. It provides a full range of chart and support classes, making it quite easy to use to produce sophisticated graphics. Because it supports PNG and SVG, it is an excellent tool for generating graphical displays for the Web using CGI scripts. You can view examples on its creator's website at *http://home.gna.org/pychart/examples/index.html*. The version supplied on that website is for Python 2. You can find a version compatible with Python 3 on this book's website, along with an example application that uses PyChart to generate dot plots through a web interface.

Example 11-2. A subclass for testing an abstract class

```
if __name__ == '__main__':

    class EmptyPlot(Plot):
        canvas_background = 'gray90'
        def get_plot_dimensions(self):
            return 200, 150, 20, 20, 10, 10
        def draw_plot(self):
            self.draw_line(0, 0, 100, 100)

    plot = None
    try:
        plot = EmptyPlot()
        plot.execute()
        input(
'''You should see a light-gray plot with a line
from (0,0) to (100,100); press Return to close ''')
    finally:
        if plot:
            plot.close()
```

> Something like this should go in almost every module. The code serves multiple purposes:
>
> - Importing or executing the module will reveal syntax errors.
> - Importing or executing the module can reveal basic problems with the implementation of the class, such as missing methods or modules.
> - The program shows the reader the basic use of the class.
> - In a graphics program, the test can display a simple diagram that shows what the class does on its own.
>
> Detecting syntax and runtime errors is not something that you should stop doing once everything seems to work: it's a process you'll want to repeat every time you make significant changes to the class.

In this case, because the class is abstract, it is not meant to be used entirely on its own. Therefore, the test code creates a very simple subclass and uses that instead. Figure 11-3 shows what it displays. Not only does this test the code, but it gives anyone developing a subclass a guide to the abstract class's basic facilities.

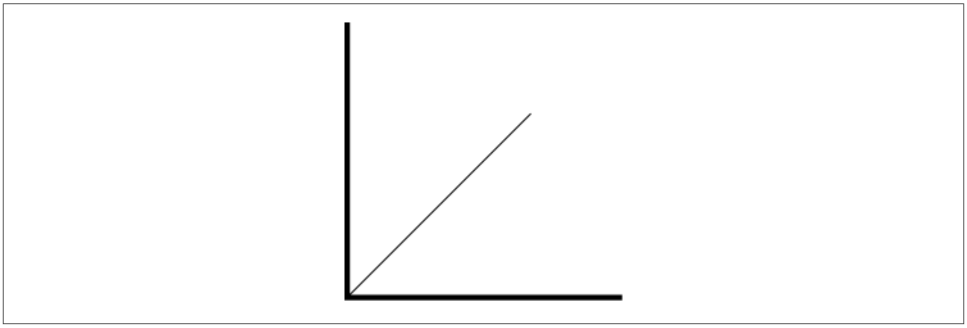


Figure 11-3. Basic example using the Plot class

The abstract class provides the following:

- Default values for class fields that control many details of the display
- An initialization method that sets off a cascade of "setup" methods that use the values defined by the class fields and the results of certain other methods
- A hierarchy of "setup" methods that defines what needs to be set up and provides reasonable defaults for some of these actions and empty methods for others
- A simplified interface to the `Canvas` drawing methods
- A few utilities, including one for finding fonts
- An `execute` method to create the widget(s), draw on the canvas, and optionally write the canvas contents out to a PostScript file

Example 11-3 shows the hierarchy of calls among the class's methods. (The actual code is available on the book's website.) What's important about this class is not its use of `tkinter`, but how it organizes everything for subclasses to more easily create certain kinds of plots using `tkinter`'s `Canvas` drawing methods.

Example 11-3. The methods of the abstract Plot class

```
Key for subclass definitions:
    x  normally would NOT implement (framework definition)
    !  MUST OVERRIDE
    *  MUST EXTEND
    +  MAY EXTEND
    /  MAY EXTEND or OVERRIDE
    -  MAY OVERRIDE (empty methods: convenience and clarity)

Utility Methods

x   NextPlotNumber()
x   closeall()
x   findfont(families, size, boldflg, italicflg)

Access
x   get_root
x   add_font
```

```
x   get_font
/   __str__

Layout Framework
*   __init__    tracks instance, creates Tk root, stores parameters
x       setup
+           setup_fonts
-           setup_data
-           setup_parameters
!           get_plot_dimensions
x           determine_layout   canvas width & height, origin x,y

Drawing Framework
x   execute
/       create_widgets
x           create_canvas
x       draw
x       draw_axes
-           draw_x_tics
-           draw_y_tics
-           draw_x_axis_labels
-           draw_y_axis_labels
!       draw_plot
x   close

Drawing Functions
x   draw_line
x   draw_line_unscaled
x   draw_oval
x   draw_rectangle
x   draw_text
x   draw_text_unscaled
```

The basic steps in a program that uses this class are:

1. Create an instance.
2. Call `execute` and wait for the user to end the program.
3. Call `close` in the `finally` clause of a `try` statement.

The subclass is required to implement only three methods:

- `__init__([`*`windowtitle`*`[, `*`scale`*`[, `*`ps_filename`*`[, `*`ps_scale`*`]]]])`

  which takes the following optional parameters:

  `windowtitle`
  : A string; if false, one will be constructed

  `scale`
  : A multiplier for all <x,y> values, including the axes

  `ps_filename`
  : The path to a file to which the contents of the canvas in PostScript can be written

ps_scale
: The scale to use for the PostScript version of the drawing

- `get_plot_dimensions()`

  which must return a tuple of the following form containing the details of the plot within the canvas:

  ```
  (width, height, leftpad, rightpad, toppad, bottompad)
  ```

  `width` and *`height`* are the dimensions of the area of the plot itself—i.e., the area defined by the extent of the x and y axes. The other values indicate how much space to leave between the corresponding edge of the canvas and the plot. These values, along with the values of certain class fields, are used in `determine_layout` to set a number of values used in organizing the display.

- `draw_plot()`

  which creates the objects that go in the `Canvas` and specifies their properties.

The subclass is free to implement `setup_data` and `setup_parameters` however it wishes. It is intended that one or both of these methods, or perhaps just the subclass's `__init__` method, will determine and store the values that will be used in `draw_plot`. Note that `draw_plot` is not responsible for drawing axis lines, tics, or labels. The drawing functions are relative to an origin at the *lower left*, with positive values above the axis. This inverts the usual direction of y coordinates in `tkinter` to make plotting more convenient.

Table 11-6 shows the class fields that control details of the display. Any of them may be overridden in a subclass. Figure 11-4 illustrates the basic layout implemented by the drawing framework. The four margins and the plot height and width are values returned by a subclass's `get_plot_dimensions` method. The two axis widths are the values of class fields. The diagonal drawn demonstrates that y values increase upward. A line like this, along with the circle and text at its end, is drawn using `Plot`'s drawing methods. Positions are expressed relative to the origin of the plot area, not the canvas.

Table 11-6. Class fields used by Plot class to control display

Name	Value	Use
`PlotName`	`'Plot'`	Combined with the current instance count to construct a default window name
`canvas_pad_x`, `canvas_pad_y`	`0`	Horizontal and vertical padding of the canvas widget itself
`canvas_border`	`0`	Border width of the canvas
`canvas_background`	`'white'`	Background color of the canvas
`x_axis_width` `y_axis_ width`	`2`	Width of the axis lines; `0` means "no line"
`x_tic_length`	`12`	Length of "tic" lines on axes if axes are drawn

Name	Value	Use
y_tic_length		
x_axis_font_size y_axis_font_size	12	Size of font for labels drawn on axis, if any
serif_faces	Lists of likely font names	Convenient defaults for subclasses to use as desired
sans_faces		
mono_faces		

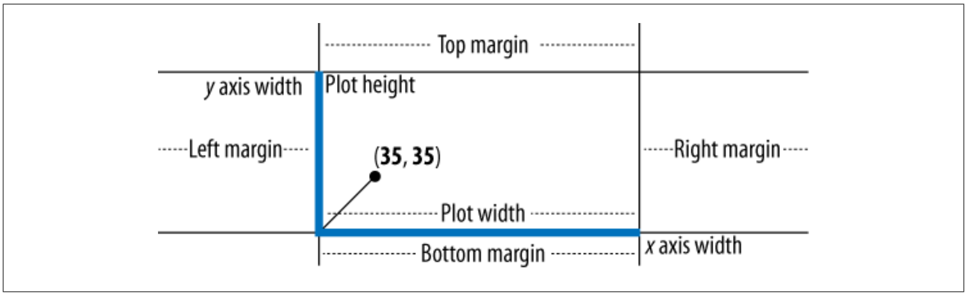


Figure 11-4. Drawing framework layout

Dot plots

Dot plots are a simple technique for visually comparing two base or amino acid sequences of similar size to highlight slight differences.§ One sequence is arranged along the horizontal axis and the other along the vertical axis. Then, the base or amino acid at each sequence on one axis is compared to every one on the other. In the simplest implementation, the plot contains a dot wherever there is an exact match. Figure 11-5 illustrates the process with two short sequences.

Due to random matches, however (more so in base sequences than in amino acid sequences), this implementation generates far too many dots to be useful for many purposes. Random matches can be reduced by using two parameters: a window `w` and a cutoff `c`. At each point, that point plus the next `w-1` points are compared, and the total number of matches is counted. A match is registered for that point only if the count is at least `c`. A graphical display of the results could use multiple levels of gray to show different scores above `c`. For some applications, amino acid sequence matching can be refined further by scoring each comparison according to a substitution matrix rather than just testing equality.

§ A good review article that covers the history, uses, and algorithms of two-sequence dot plots in bioinformatics can be found at *http://www.code10.info/index.php?view=article&id=64*. It includes explanations of how the various kinds of differences between a pair of sequences appear in dot plots.

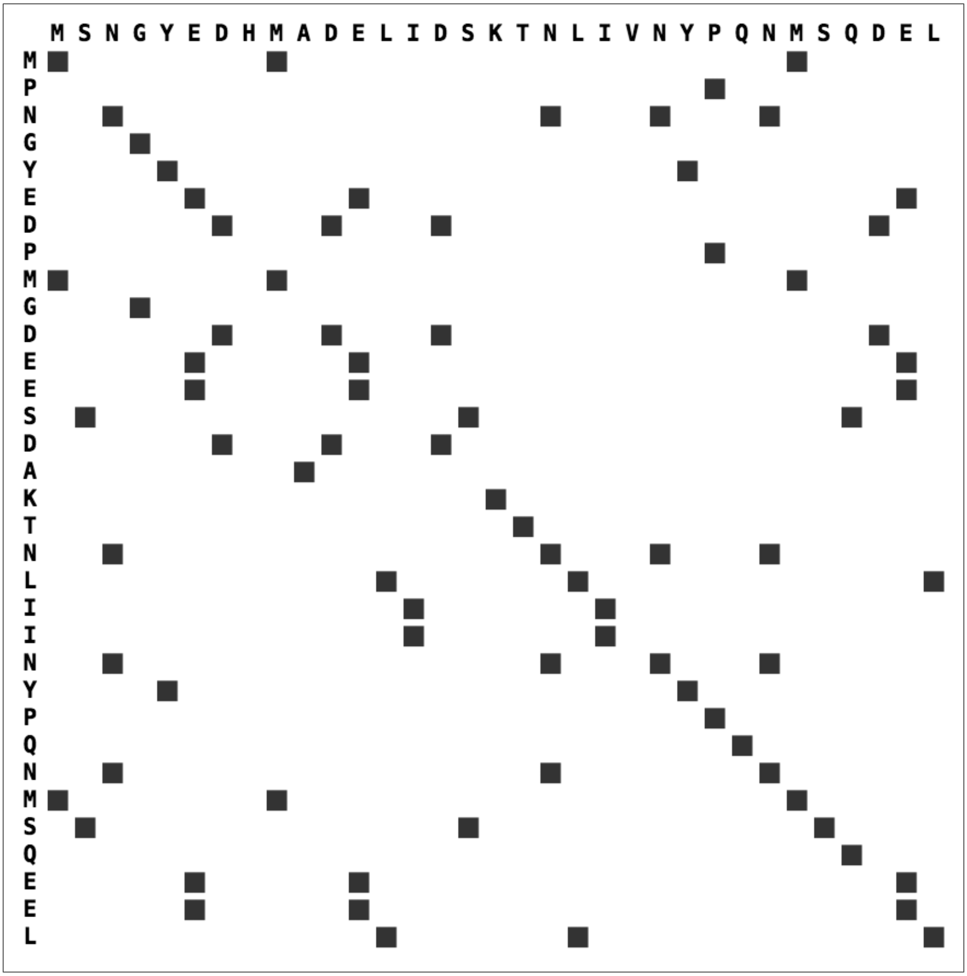


Figure 11-5. Comparison of two sequences for a dot plot

Figure 11-6 illustrates the effects of these parameters for a short amino acid sequence compared against itself. On the left, `w` and `t` are both `1`. On the right, `w=3` and `t=2`. These are more typical values for amino acid sequences (typical values for base sequences would be `w=11` and `c=7`). The actual values to use for these parameters will depend on the purpose of the comparison. An application could easily provide controls that allow the user to see what the dot plot looks like for different values of `w` and `t`.

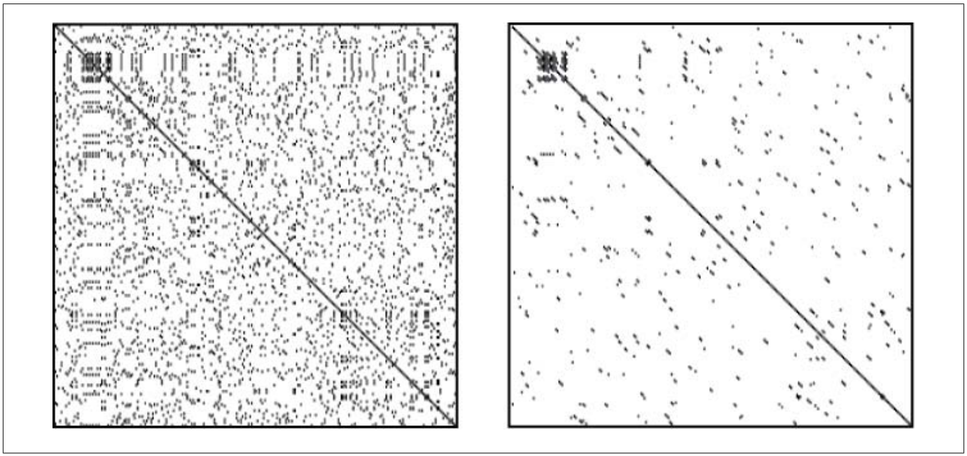


Figure 11-6. A dot plot with different windows and cutoffs

A perfect match between two sequences will show a solid diagonal line. Two sequences that match exactly except for their initial parts will also show a solid diagonal line, but unless the nonmatching parts are the same length the diagonal will not be along the x=y diagonal of the graph. Dots scattered elsewhere are not meaningful unless clustered in certain ways.

Dot plots can visually highlight evolutionary modifications both between two similar sequences and within a sequence. The effects of the possible changes are listed in Table 11-7. Some applications display dot plots with the diagonal going from upper-left to lower-right, and others from lower-right to lower-left. For this reason, the table just says "toward the right" or "toward the left." The direction is to be interpreted as relative to the x=y diagonal.

Table 11-7. Appearance of mismatches in dot plots

Change in y-axis sequence	Visual effect
Mutation	Small gaps in the line
Deletion	Line offset to the right of the diagonal
Insertion	Line offset to the left of the diagonal
Duplication	Line offset to the left of the diagonal, with a parallel overlap corresponding to the duplication
Inversion	Large gap with a faint, broken line of similar length perpendicular to the gap and centered on its center
Self-duplication	Two line segments reflected around the diagonal when the sequence is compared to itself

Two other patterns have special significance. One is an artifact of sequencing technology: a stretch of Ns in one sequence where the other sequence has real bases will result in a horizontal or vertical swatch without dots.

The other pattern does have biological significance, but not necessarily an evolutionary one: an area of low complexity manifests itself as dense, perhaps even solid, squares around the diagonal. In base sequences, these squares reveal areas with many repeats of a very short subsequence. An amino acid sequence would usually not have multiple repeats of exactly the same short subsequence, but a stretch containing only a few of the possible amino acids would have a similar effect (e.g., `EKLKEKEKQKEKERQREKEK`). These squares can even reveal regions of *self-similarity* when a sequence is compared with itself.

Figures 11-7, 11-8, and 11-9 illustrate the way the changes shown in Table 11-7 appear in dot plots. These illustrations were constructed manually by making changes in a copy of the x-axis sequence, an amino acid sequence of length 292. (Dot plots are often used for much larger sequences, of course, such as cDNA sequences.)

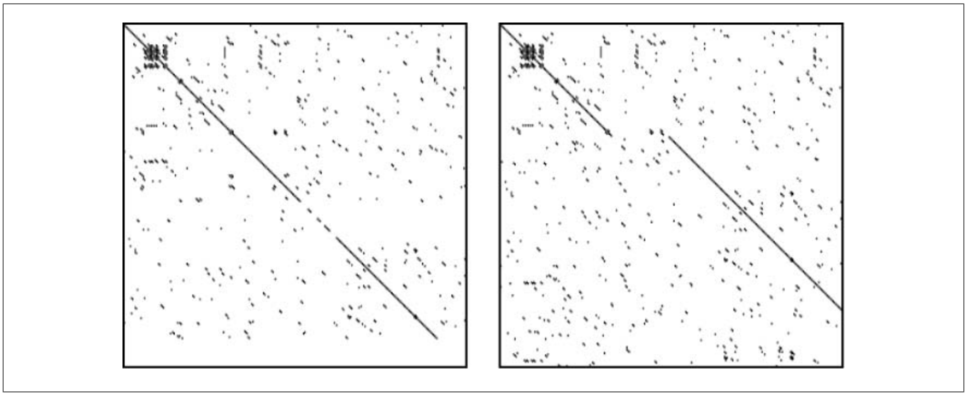


Figure 11-7. Mutations and deletions in a sequence dot plot

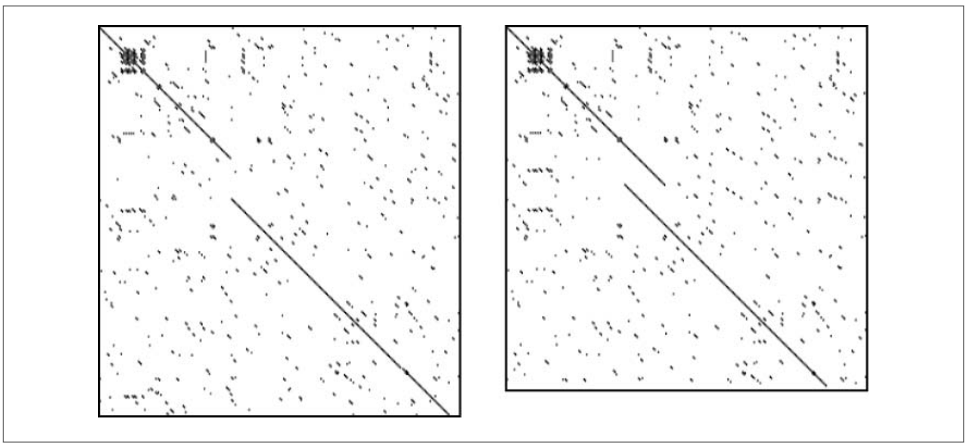


Figure 11-8. Insertion and duplication in a sequence dot plot

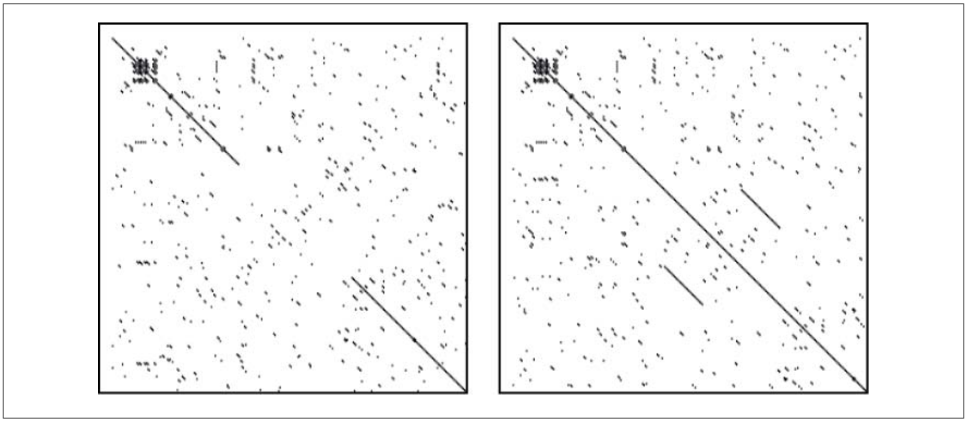


Figure 11-9. Inversion and self-duplication in a sequence dot plot

Next, we'll look at a complete definition of a class for generating dot plots. It is based on the abstract `Plot` class described earlier. Where methods are defined that override or extend the inherited versions, the method name is highlighted. In addition, if the method extends the inherited method, the call to `super` is highlighted.

In anticipation of its use in real applications, our class provides a class method that returns an `optparser` facility. One of its `__init__` method's required arguments is an `OptionParser` that has already parsed the command line. Quite a few parameters are set through the parser. Rather than passing them separately when the `DotPlot` is created, the option object packages them. For convenience, methods throughout the class are defined to access fields of the `OptionParser` pretty much as if they had been fields of the `DotPlot` object.

The module includes test code just the way the abstract class's module did, as shown in Example 11-4. It generates a random sequence 201 amino acids long and plots it against itself. Figure 11-10 shows the plot displayed.

Example 11-4. Test code for the DotPlot module

```
from random import randint

aacodes = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M',
           'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'X', 'Y', 'Z']

def generate_sequence(length):
    return ''.join([aacodes[randint(0, len(aacodes)-1)]
                   for n in range(length)])

if __name__ == '__main__':
    string = generate_sequence(202)
    plot = DotPlot(string, string, window=3, threshold=2, with_axes=True)
    plot.execute()                                      # defined in Plot
    try:
        sys.ps1                                         # are we running interactively?
```

```
    except:                                          # no
        input("Press the Return key to close the window(s)")
        plot.close()
```

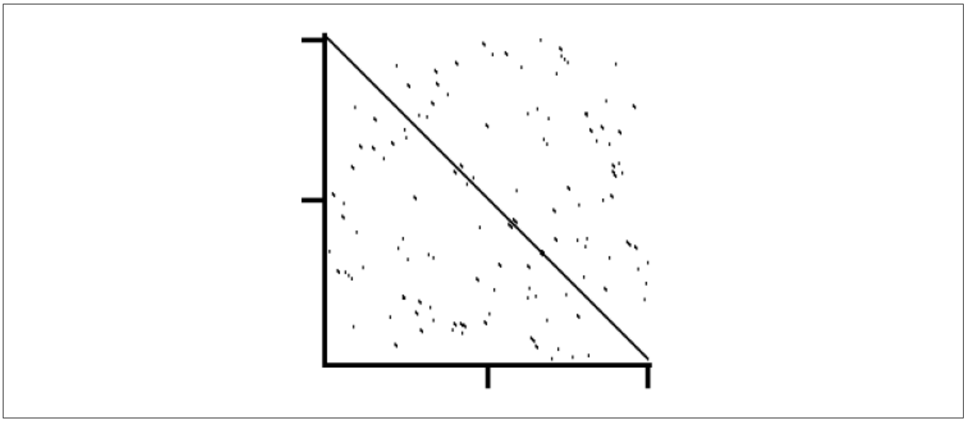


Figure 11-10. The plot drawn by the DotPlot test code

Example 11-5 is the first part of the class definition. It overrides a couple of class fields and defines similar parameters for its own use. The `__init__` method it shows is unusual in that it calls its superclass's `__init__` *last* rather than first. This is because the abstract class's `__init__` method sets off a cascade of initialization and setup methods that will use some of the values stored by this class's `__init__` method.

Example 11-5. Generating a dot plot using tkinter

```
class DotPlot(Plot):

# Overridden class fields
    x_tic_width = y_tic_width = 3
    sans_faces = ('Helvetica Narrow', 'Futura',
                  'Helvetica', 'Arial', 'sans-serif')
    PlotName = 'Dot Plot'

# Fields specific to this class
    title_font_size = 12
    top_title_margin = round(title_font_size / 2)
    name_height = 22
    tic_width = 3
    axis_width = 3
    # Defaults for showing the plot without axes;
    # if options.axes is true, they will be reset in setup_parameters
    left_title_margin = 10
    xaxis_width = yaxis_width = 0
    xlabel_height = ylabel_width = 0

    def __init__(self, seq1, seq2, seqname1 = '', seqname2='',
                 window=1, threshold=1, with_axes=False, dot_size=1,
                 # super parameters:
```

```
                 window_title=None,
                 scale=1.0, ps_filename=None, ps_scale = 1.0):
        self.seq1 = seq1
        self.seq2 = seq2
        self.seqname1 = seqname1
        self.seqname2 = seqname2
        self.window = window
        self.threshold = threshold
        self.with_axes = with_axes
        self.dot_size = dot_size
        self.window_title = window_title
        super().__init__(options.name, scale, ps_filename, ps_scale)
```

Example 11-6 contains configuration methods that extend or replace definitions inherited from `Plot`. These mostly determine details that are specific to the kind of plot the class implements.

Example 11-6. Plot setup methods overridden in DotPlot

```
    def setup_fonts(self):
        super().setup_fonts()
        self.add_font('title', self.findfont(self.sans_faces,
                                             self.title_font_size))
    def setup_data(self):
        self.points = self.compute_points()
        self.max_x = max(self.points, key=lambda pt: pt[0])[0]
        self.max_y = max(self.points, key=lambda pt: pt[1])[1]

    def setup_parameters(self):
        super().setup_parameters()
        if not self.with_axes:
            self.x_axis_width = self.y_axis_width = 0
        else:
            self.x_axis_width = self.axis_width
            self.y_axis_width = self.axis_width
            self.x_label_height = self.x_tic_length + 5
            self.y_label_width = self.y_tic_length + 5
            self.left_title_margin = \
                self.left_title_margin + self.y_label_width

    def get_plot_dimensions(self):                # required
        return  (round((self.max_x + self.window) * self.scale),
                 round((self.max_y + self.window) * self.scale) +
                 self.y_tic_width,
                 self.y_label_width,
                 self.ipad_right,
                 self.name_height *
                 (bool(self.seqname1) + bool(self.seqname2)),
                 self.x_label_height)
```

Example 11-7 shows the code that determines the points to be plotted. The `setup_data` method from Example 11-6 calls `compute_points` and assigns the `points` field to its result.

Example 11-7. Computing the points to be drawn in a DotPlot

```
def compute_points(self):
    pts = []
    for y in range(1 + len(self.seq2) - self.window):
        for x in range(1 + len(self.seq1) - self.window):
            if self.test_point(self.seq1, x, self.seq2, y):
                pts.append((x, y))
    return pts

def test_point(self, seq1, x, seq2, y):
    cnt = 0
    for n in range(self.window):
        if seq1[x+n] == seq2[y+n]:
            cnt += 1
    return cnt >= self.threshold
```

Example 11-8 implements the class's drawing operations. If sequence names were provided in the call to `DotPlot`, their names are drawn above the plot. If the `with_axes` argument is false, the `setup_parameters` method from Example 11-6 sets the width of both axes to `0`, and `Plot` does not draw the axes. If it's true, `setup_parameters` sets the axis widths along with some other parameters. In that case, the superclass draws the axis lines and calls `draw_x_axis_labels` and `draw_y_axis_labels`. As defined, these functions just draw tic marks every 100 points along each axis. Finally, a circle (oval) of diameter `dot_size` pixels is drawn for each of the computed points. Few graphical user interface facilities actually provide a point-drawing operation; instead, programs simply draw very small circles.

Example 11-8. DotPlot's drawing methods

```
def draw(self):
    super().draw()
    self.draw_titles()

def draw_titles(self):
    if self.seqname1 and self.seqname2:
        self.draw_title('x = ' + self.seqname1,
                        self.top_title_margin)
        self.draw_title('y = ' + self.seqname2,
                        self.top_title_margin +
                        self.title_font_size + 6)
    elif self.seqname1 or self.seqname2:
        self.draw_title(self.seqname1 or self.seqname2,
                        self.top_title_margin)

def draw_title(self, title, ypos):
    self.draw_text_unscaled(self.left_title_margin,
                            ypos, title, 'title', 'nw')

def draw_x_axis_labels(self):
    for x in range(100, self.plot_width+1, 100):
        self.draw_line(x, -self.x_axis_width,
                       x,  -(self.x_axis_width + self.x_tic_length),
```

```
                              self.x_tic_width)

    def draw_y_axis_labels(self):
        adjust = round((self.y_tic_width - 1)/2)
        for y in range(100, self.plot_height+1, 100):
            self.draw_line(-self.y_axis_width,
                              y + adjust,
                              -(self.y_axis_width + self.y_tic_length),
                              y + adjust,
                              self.y_tic_width)

    def draw_plot(self):                              # required
       for pt in self.points:
            self.draw_oval(
                pt[0],
                self.plot_height - self.window - pt[1],
                pt[0] + self.dot_size - 1,
                self.plot_height - self.window - pt[1] - self.dot_size - 1)
```

Histograms

An interesting use of histograms is to plot the distribution of codon usage among the known coding sequences for an organism. The program shown in this section color-codes the bars according to the GC content of the corresponding codon. The GC content for all of the cDNA sequences from which the data was taken is written at the upper left of the graph. (Note that this is not the same as the GC content for the organism's DNA as a whole, because it is based on coding sequences.) The code to extract the data from a file is not shown here, but it is quite similar to our many FASTA examples.‖

Histograms for two different bacteria are shown in Figures 11-11 and 11-12. Looking at the two figures, it's easy to see that the distributions of codons used by these two species are substantially different.#

Example 11-9 shows the class fields of another subclass that uses the abstract `Plot` class shown earlier. Method definitions are shown in subsequent examples. In those examples, the names of methods that override or extend methods inherited from `Plot` are highlighted, as are calls to `super`.

‖ The program's data source is a file of bacterial species codon distributions downloaded from *ftp://ftp.kazusa.or.jp/pub/codon/current/gbbct.codon*. Interesting interactive tools for getting codon counts in several formats for species matching a regular expression can be found at *http://www.kazusa.or.jp/codon*.

The downloaded data at the previously mentioned site was computed from the GenBank flat file release of June 15, 2007, using the cDNAs available for each species. The datafile simply contains alternating lines of species information and 64 integers representing counts per thousand of each codon. The program doesn't do anything to compute those values; it just graphs them.

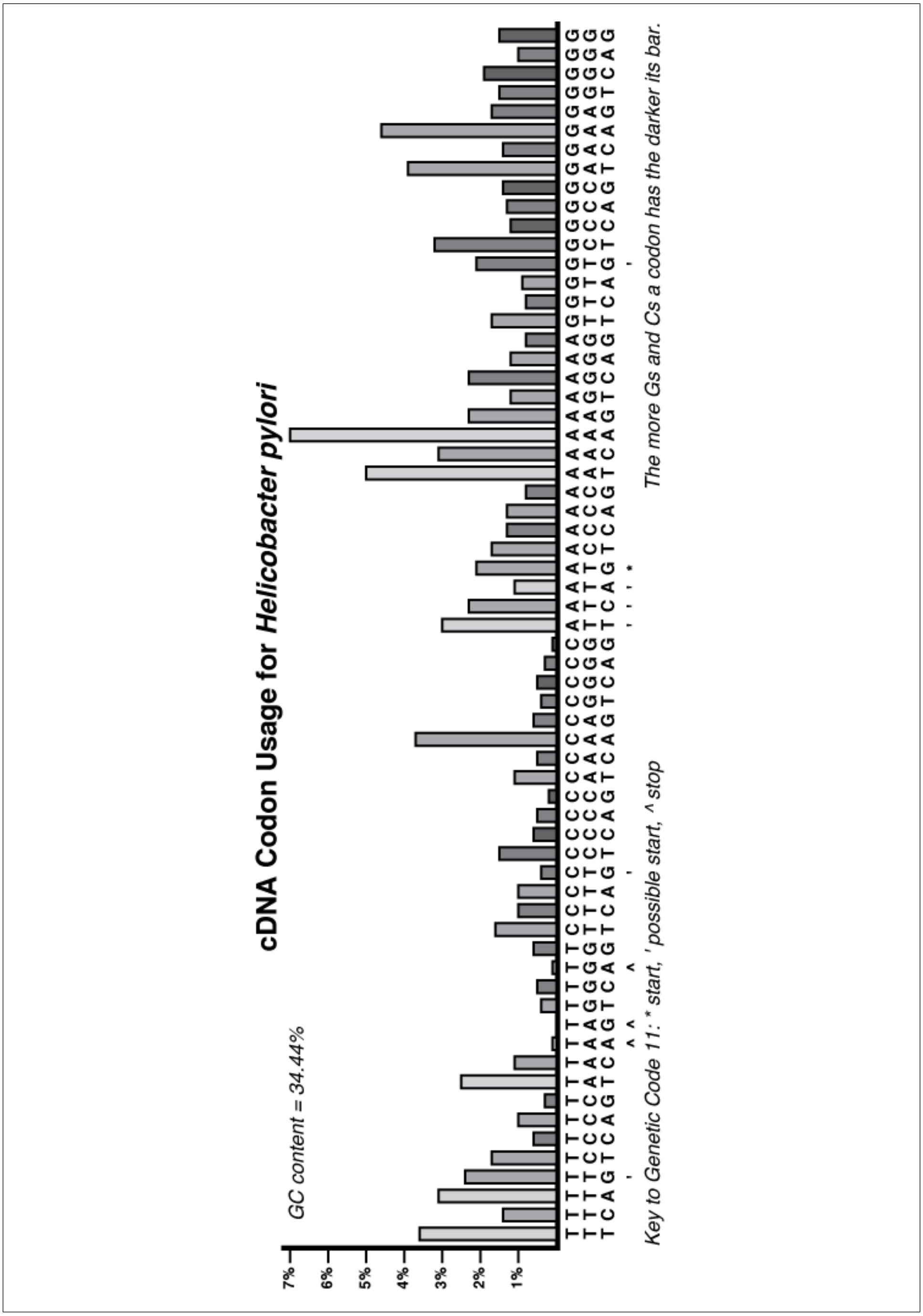


Figure 11-11. Codon usage histogram for Helico pylori

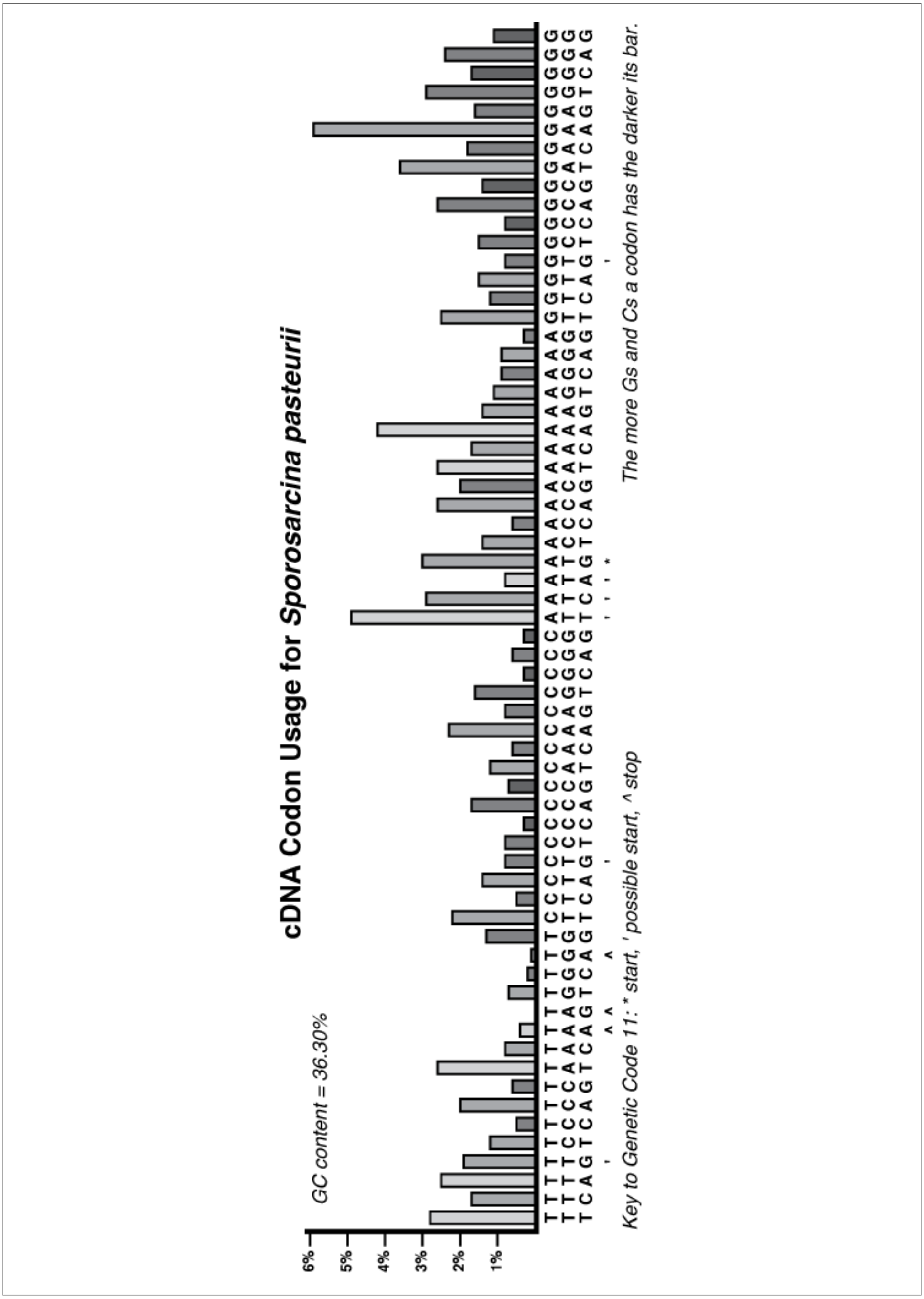


Figure 11-12. Codon usage for Sporosarcina pasteurii using tkinter

Example 11-9. Codon distribution histograms using tkinter

```
class CodonUsageHistogram(Plot):

# Fields that override inherited values
    x_tic_length = y_tic_length = 16

# Fields specific to this class
    barwidth = 10                                    # width of histogram bars
    barspacing = 5                                   # spacing of histogram bars
    barscale = 3                                     # vertical pixels per data point
    codon_font_size = 14
    marker_font_size = 14
    title_font_size = 22
    key_font_size = 16
    codon_faces = ('Liberation Mono Regular',
                   'Lucida Sans Typerwriter',
                   'DejaVu Sans Mono',
                   'Bitstream Sans Mono',
                   'Courier')
    gc_colors = ('gray80', 'gray60', 'gray40', 'gray25')
    default_bar_color = 'gray94'
    plot_top_margin = 20
    ylabel_width = 60
    x_label_height = codon_font_size*5
    canvas_background = 'white'
    title_pad_y = 6
    key_pad_y = 12

    codons = [base1 + base2 + base3
              for base1 in 'TCAG'
              for base2 in 'TCAG'
           for base3 in 'TCAG']
    codon_labels = codons[:]                         # a copy to modify
    startchar = '*'
    altstartchar = "'"
    stopchar = '^'
    gc_colors = ('gray80', 'gray60', 'gray40', 'gray25')
    # add start and stop indicators
    # stop codons:
    for codon in ('TAA', 'TAG', 'TGA'):
        codon_labels[codon_labels.index(codon)] += stopchar
    # standard start codon:
    codon_labels[codon_labels.index('ATG')] = 'ATG' + startchar
    # alternate start codons in code 11
    for codon in ('TTG', 'CTG', 'GTG', 'ATT', 'ATA', 'ATC'):
        codon_labels[codon_labels.index(codon)] += altstartchar
```

Notice the `for` statements in the class field section. Remember that when a `class` statement is executed, each of the statements it contains is executed. Although normally classes contain just assignment and `def` statements, they can contain loops, iterations, conditionals, and so on. These `for` statements and associated assignments set up a list of codon labels that incorporate start and stop codons as well as the alternate start codons for the bacterial genetic code.* (The class could easily be generalized to allow specifying a code as an argument when creating an instance; what's shown here assumes bacterial sequences.)

Example 11-10 shows the definition of `__init__`. The `name` parameter is the name of the organism, for the purpose of displaying a title. The data is simply a dictionary or list of pairs that gives the count per thousand of the corresponding codon. The other methods in the example are part of the setup area of the framework.

Example 11-10. Setup methods for the tkinter codon distribution plot

```
def __init__(self, name, data,
             # super paramaters:
             windowtitle=None, scale=1.0,
             ps_filename=None,  ps_scale = 1.0):
    self.data = dict(data)
    self.name = name
    super().__init__(name, scale, filename, ps_scale)

def setup_fonts(self):
    super().setup_fonts()
    self.add_font('x', self.findfont(self.codon_faces,
                                     self.codon_font_size,
                                     True))
    self.add_font('title', self.findfont(self.sans_faces,
                                         self.title_font_size,
                                         True))
    self.add_font('key', self.findfont(self.sans_faces,
                                       self.key_font_size,
                                       False, True))
    self.add_font('marker', self.get_font('x').copy())
    self.get_font('marker').configure(size=self.marker_font_size)
    self.add_font('species', self.get_font('title').copy())
    self.get_font('species').configure(slant='italic')

def setup_data(self):
    self.maxval = max(self.data.values())
    self.sumval = sum(self.data.values())
    self.maxpercent = round((100 * self.maxval) / self.sumval)

def get_plot_dimensions(self):                          # required
    return (self.barspacing + round(64 * self.scale *
```

* All the alternate start codons documented for the bacterial genetic code at *http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi#SG11* are included, even those found very rarely.

```
                                (self.barwidth + self.barspacing)),
            round(self.scale * self.barscale * (2 + self.maxval)),
            self.ylabel_width + 20,
            20,
            self.get_title_height(),
            self.xlabel_height + self.get_key_height()
            )

    def get_title_height(self):
        return 2 * self.title_font_size

    def get_key_height(self):
        return 2 * self.key_font_size
```

Example 11-11 shows the definitions of the methods that override the empty superclass methods for drawing the tics and labels of the y-axis and the elaborate codon labels along the x-axis.

Example 11-11. Axis methods for the tkinter codon distribution plot

```
    def draw_y_tics(self):
        for n in range(1, self.maxpercent + 1):
            vertpos = 10 * n * self.barscale
            # * 10 because barheights are per thousand
            self.draw_line(0, vertpos, -self.y_tic_length, vertpos)

    def draw_y_axis_labels(self):
        for n in range(1, self.maxpercent + 1):
            self.draw_text(-self.tic_length-4,
                           10 * n * self.barscale,
                           str(n) + '%',
                           'y',
                           anchor='e')

    def draw_x_axis_labels(self):
        curx = self.barspacing + 5
        for n, codon in enumerate(self.codon_labels):
            self.draw_codon_label(curx, codon),
            curx += self.barwidth + self.barspacing

    def draw_codon_label(self, x, codon):
        for n, char in enumerate(codon):
            self.draw_text(x,
                          # 6 * is additional separation for markers
                           -((self.codon_font_size * (n+1)) +
                             (6 * (n > f2))),
                          char,
                          'x' if n < 3 else 'marker',
                          anchor='center')
```

The methods in Example 11-12 draw the title above the plot, the key below, and an indication of the total GC content of the coding sequences at the upper left of the plot area. The computation of the total GC content is done by the `gccontent` method shown in the subsequent example.

Example 11-12. Title and key methods for the distribution plot

```
def draw(self):
    super().draw()
    self.draw_title()
    self.draw_gc_content()
    self.draw_key()

def draw_title(self):
    self.draw_text(round(self.plot_width / 2), self.plot_height,
                   'cDNA Codon Usage for ',  'title',  anchor='se')
    self.draw_text(round(self.plot_width / 2), self.plot_height,
                   self.name, 'species',  anchor='sw')

def draw_gc_content(self):
    self.draw_text(20, self.plot_height-20,
                   'GC content = {:.2f}%'.format(self.gccontent()),
                   'key', 'sw')

def draw_key(self):
    self.draw_text(
        0,  -self.xlabel_height,
        'Key to Genetic Code 11: ' +
        '{} start, {} possible start, {} stop'.
        format(self.startchar, self.altstartchar, self.stopchar),
        'key',  anchor='nw')
    self.draw_text(
        self.plot_width, -self.xlabel_height,
        'The more Gs and Cs a codon has the darker its bar.',
        'key', anchor='ne')
```

The methods in Example 11-13 draw the usage bars. Each bar is drawn in one of four shades of gray, according to the number of Gs and Cs in the codon. The two class methods are class methods simply because nothing they do is specific to the instance. The `gccontent` method calls `codon_gc_count` as part of its computation of the total GC content for the sequences. That is not a class method, because its computation depends on the instance's `data` field.

Example 11-13. Methods for drawing the distribution bars

```
@classmethod
def codon_gc_count(self, codon):
    return codon.count('G') + codon.count('C')

@classmethod
def gccolor(self, codon):
    return self.gc_colors[self.codon_gc_count(codon)]

def gccontent(self):
    # 3 because there are 3 bases in each codon
    # 10 because the counts are per thousand
    return (3 * 10 * sum((self.codon_gc_count(codon) * count
                          for codon, count in self.data.items())) /
```

```
            self.sumval
            )

    def draw_plot(self):                            # required
        curx = self.barspacing
        for codon in self.codons:
            n = self.data[codon]
            self.draw_rectangle(curx,  0,  self.barwidth,  n * self.barscale,
                                self.gccolor(codon))
            curx += self.barwidth + self.barspacing
```

The code in this chapter has a very different "shape" than other Python code shown in this book. While the overall organization of graphics programs can make good use of object-oriented techniques, in the end drawing comes down to a series of commands to draw specific things with specific properties in specific places. This results in many procedural-style function calls with many arguments.

You might also have noticed some funny arithmetic going on: adjustment factors, adding 1 or 2, calls to `round`, and so on. These are not typically manipulations you can anticipate, and they won't necessarily be the same if you implement the same program with a different toolkit. Rather, they come from seeing the results of straightforward calculations and then adjusting them. It is always best to at least name such values in a way that indicates their purpose, such as `adjustment` or `border-displacement`. Comments are more useful in this kind of programming than they are in more straightforward code.

Structured Graphics with SVG

SVG—*Scalable Vector Graphics*—defines a set of XML markup tags and attributes for describing shapes and text to be displayed in a web browser.† The name of the technology emphasizes several important characteristics:

Vector- versus pixel-based graphics
: Most image formats represent pixels. They contain no information about what the pixels represent, so transformations such as zooming in and out can operate only on pixels. In contrast, a vector-based system *describes* what is in the image. The description is used to generate pixels when the image is displayed. (This is analogous to the difference between representing fonts as bitmaps versus representing them as unit-independent curves.)

† See *http://www.w3.org/TR/SVG11* for the complete specification. Much of it is very formal and sophisticated, but the sections on shapes (*http://www.w3.org/TR/SVG11/shapes.html*) and text (*http://www.w3.org/TR/SVG11/text.html*) are very useful references for even basic use of SVG. Both have excellent examples, including the resulting images, and would constitute an excellent supplement to the brief descriptions in this chapter.

Scalable technology

The term "scalable" in SVG means two things. The first one, which we won't talk about here although it's important in other contexts, is that the technology itself is scalable: one SVG file can refer to many others to build highly complex images and can be used by many different kinds of applications.

Scalable graphics

What interests us here is that a vector-based image is in itself "dimensionless." The actual image is generated for a specified resolution. This means the images can be zoomed in and out and used at different sizes on different web pages. A scalable representation supports the different requirements of different output media—i.e., the widely varying resolutions of different makes and models of displays and printers.

SVG is quite powerful: it supports scaling, resizing, interaction, and animation. An SVG file can include Cascading Style Sheets (CSS) and scripts (JavaScript). SVG content can be delivered to a browser in several forms, including:

- In a file containing just SVG
- By URL reference to a separate SVG file in an HTML or XHTML file `<object>` or `<img>` tag
- Directly inside an XHTML file

For simple situations, a self-contained SVG file is fine. If the image is only a part of a web page to be displayed, using an `object` or `img` tag is appropriate. More sophisticated client-side programming techniques are best supported by embedding the SVG content inside an XHTML file. We will use a plain SVG file here.

SVG is generally well supported by modern browsers. In the same way that a browser interprets HTML to display a web page and other graphics formats—such as PNG, JPEG, or GIF—to display an image, it interprets the contents of an SVG file and generates the corresponding image to display. A browser's zoom commands should zoom in and out on the image, and if the entire image doesn't fit in the window the browser should show scroll bars. (Actual behavior varies among browsers.)

SVG File Contents

The SVG File template shows the general outline for an SVG file.

> The "standalone" on the first line means the file refers to outside documents, which it does in the first few lines. It does *not* mean that this is an independent SVG file as opposed to one referenced by or embedded in another web page.

The `width` and `height` parameters can be specified as pixels (e.g., `400`) or percentages followed by a percent sign (e.g., `100%`). There are a number of other possibilities as well, but those will serve our purposes.

The template shows where stylesheet information and the actual SVG tags belong. We'll see examples of these a little further on in the chapter.

TEMPLATE

SVG File

The outline of an SVG file is:

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns='http://www.w3.org/2000/svg' version='1.1'
     width='width'  height='height'>
  <defs>
    <style type='text/css'><![CDATA[
       ... CSS style rules here, just as in HTML ...
      ]]>
    </style>
  </defs>
  ... SVG tags ...
</svg>
```

SVG tags

In its complete form, SVG is a sophisticated markup language. For many purposes, though, all that's needed is a simple repertoire of tags and a few of their more common attributes. Tables 11-8 (shapes) and 11-9 (text) describe those. In addition, all of the tags support the `style` attribute, which is described next. In the following, note that y coordinates start with `0` at the top of the diagram and increase downward. The default for all unspecified coordinates is `0`.

Table 11-8. SVG shape tags

Tag	Attributes	Meaning
`rect`	`x`	Smaller of the two x-axis coordinates
	`y`	Smaller of the two y-axis coordinates
	`width`	A positive value
	`height`	A positive value
	`rx`	x-axis radius of rounded corners
	`ry`	y-axis radius of rounded corners
`circle`	`cx`	x-axis coordinate of center

Tag	Attributes	Meaning
	`cy`	y-axis coordinate of center
	`r`	Radius
`ellipse`	`cx`	x-axis coordinate of center
	`cy`	y-axis coordinate of center
	`rx`	x-axis radius
	`ry`	x-axis radius
`line`	`x1`	x-axis coordinate of starting point
	`y1`	y-axis coordinate of starting point
	`x2`	x-axis coordinate of ending point
	`y2`	y-axis coordinate of ending point
`polyline` (open)	`points`	Quoted list of whitespace-separated points, each a comma-separated x,y pair
`polygon` (closed)	`points`	Quoted list of whitespace-separated points, each a comma-separated x,y pair

Table 11-9. SVG text tag

Tag	Attributes	Meaning
`text`	`x`	x-axis coordinate of start of text
	`y`	y-axis coordinate of start of text
	`width`	Width of text box (optional)
	`height`	Height of text box (optional)

SVG styles

SVG tags can be styled using CSS, just the way HTML tags can be. Two attributes, shown in Table 11-10, can be used to specify a tag's style. In addition, styles can be defined within the `style` tag in the `defs` section at the beginning of an SVG document.‡

Table 11-10. SVG style attributes

Attribute	Effect	Value types
`style`	Specifies the style for the tag	Semicolon-separated *`name`*`:`*`value`* pairs
`class`	Applies the style for the class as defined in the `style` section	A name
`id`	Applies the style for the elements with this `id` as defined in the `style` section	A name

‡ Full details regarding the use of CSS with SVG can be found at *http://www.w3.org/TR/SVG11/styling.html#StylingWithCSS*.

The basic style properties are described in Table 11-11. Table 11-12 shows the properties used to specify fonts.[§]

Table 11-11. Major SVG style properties

Property	Effect	Value type
`fill`	Inside color	Color (name[a], `#rgb`, etc.)
`stroke`	Border color	Color (name, `#rgb`, etc.)
`stroke-width`	Width of line	Number
`transform`	Modifies element (explained in following discussion)	Space- or comma-separated transforms in function-like notation

[a] See *http://www.w3.org/TR/SVG/types.html#ColorKeywords* for a list of acceptable color names and their RGB equivalents.

Table 11-12. SVG font properties

Property	Effect	Value type
`font-family`	Typeface	Comma-separated list of font names to try, in the order in which they appear. Names containing whitespace must be quoted. `serif`, `sans-serif`, and `monospace` refer to the corresponding browser settings.
`font-size`	Size	Number representing points.
`font-style`	Italicization	Normal, italic, or oblique.
`font-weight`	Boldness	Normal, bold, bolder, or lighter.

The transformation sublanguage provides a powerful range of capabilities for modifying shapes and text. These include the following:[‖]

`translate(`*`tx`*` [`*`ty`*`])`
: Add *`tx`* to the element's x position and *`ty`* (`0` if omitted) to its y position.

`scale(`*`sx`*` [`*`sy`*`])`
: Scale the element by a factor of *`sx`* horizontally and *`sy`* (*`sx`* if omitted) vertically.

`rotate(`*`angle`*` [`*`cx cy`*`])`
: Rotate by *`angle`* around the point `<`*`cx, cy`*`>` (`<0, 0>` if omitted, which usually has very surprising effects, if you even see the object after it has been rotated).

In addition to the tags for shapes, paths, and text, SVG provides a grouping tag, `g`. This is similar to the `div` tag in HTML. It provides a way to apply styles and transformations to all the tags between the opening `<g>` and the closing `</g>`. The style for the group can be specified in all the usual ways: as the value of a `style` attribute, with a `class` attribute that corresponds to a style class, or with an `id` that corresponds to a style defined for that ID.

§ See *http://www.w3.org/TR/2008/REC-CSS2-20080411/fonts.html#font-specification* for full details.

‖ *http://www.w3.org/TR/SVG11/coords.html#TransformAttribute* contains the full documentation.

Examples

Next, we'll look at two examples of programs that produce structured graphics using SVG. The first repeats the `tkinter` histogram example without the benefit of a graphics library. Because SVG files are ultimately just XML text, it is straightforward to generate them with ordinary Python code. For the same reason, though, SVG is less useful for graphics composed primarily of large numbers of individual points, such as the dot plots we saw earlier. Each point would require a separate SVG tag, and there could be tens of thousands of points in a real dot plot.

Histograms of codon use

Example 11-14 shows the outline of a program for producing the same sort of histograms as in the earlier `tkinter` examples. Example 11-15 shows the resulting SVG text. The code isn't shown here, since it is somewhat long and doesn't contain anything we haven't already seen; it is, however, available on the book's website.

Example 11-14. The outline of the codon usage histogram program

```
chart_usage
    draw_chart
        draw_heading
        draw_key
            draw_text
        draw_title
            draw_text
        draw_axes
            draw_line
            draw_text
        draw_codon_label
        draw_bars
            draw_bar
        draw_closing
```

Figures 11-13 and 11-14 correspond to Figures 11-12 and 11-11. There are slight differences, largely because of the way the x-axis was labeled with codons. In the `tkinter` implementation, separate calls to `Canvas.create_text` were used to stack characters vertically. Using SVG would have been fairly awkward, but SVG's transforms made it easy to rotate the text.

> Although browser support for SVG has come a long way, browsers don't generally implement every possible feature and sometimes don't get the features they do implement quite right. It would be worth viewing the SVG files for this example in more than one browser.

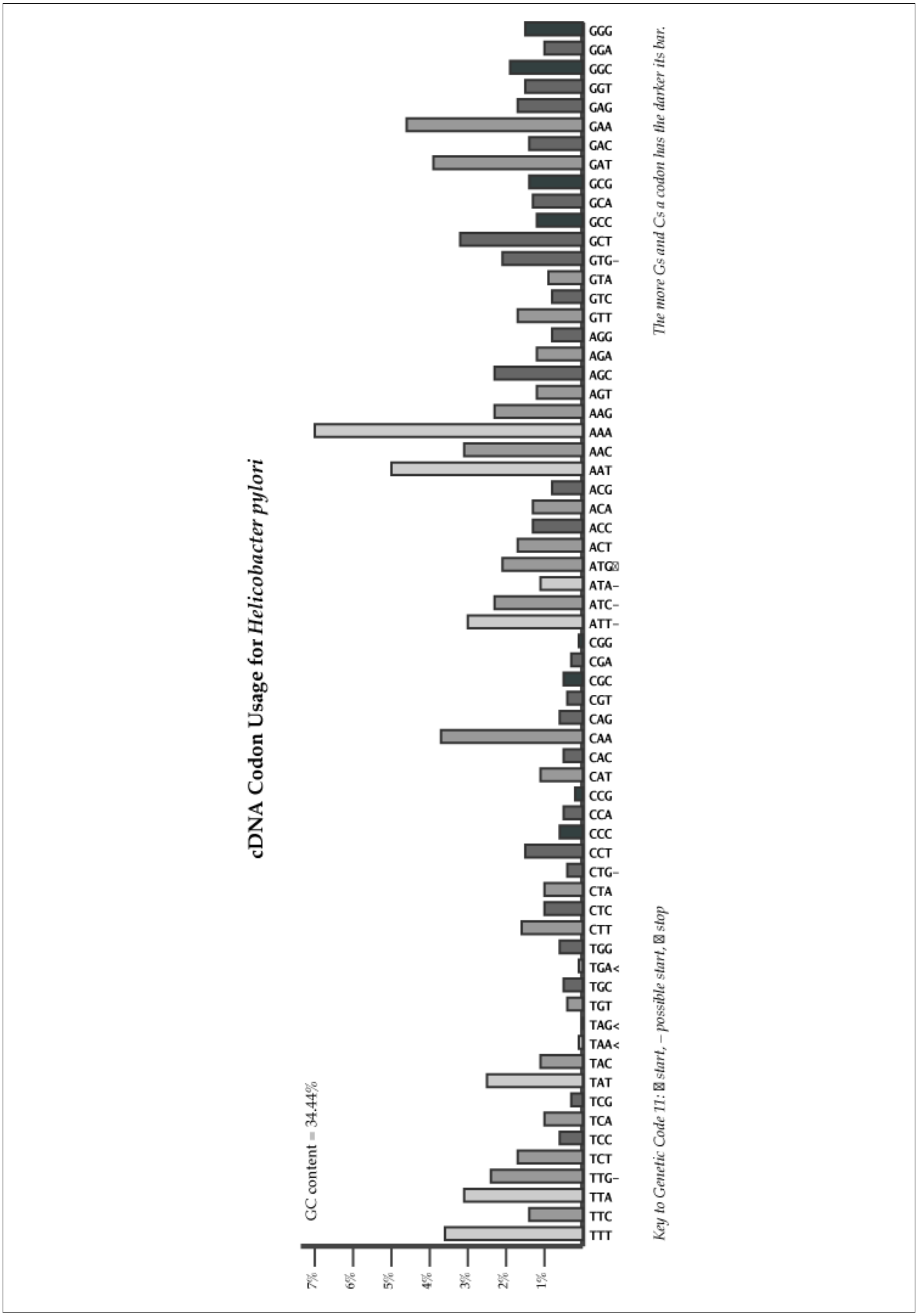


Figure 11-13. Chart of Helicobacter pylori codon usage

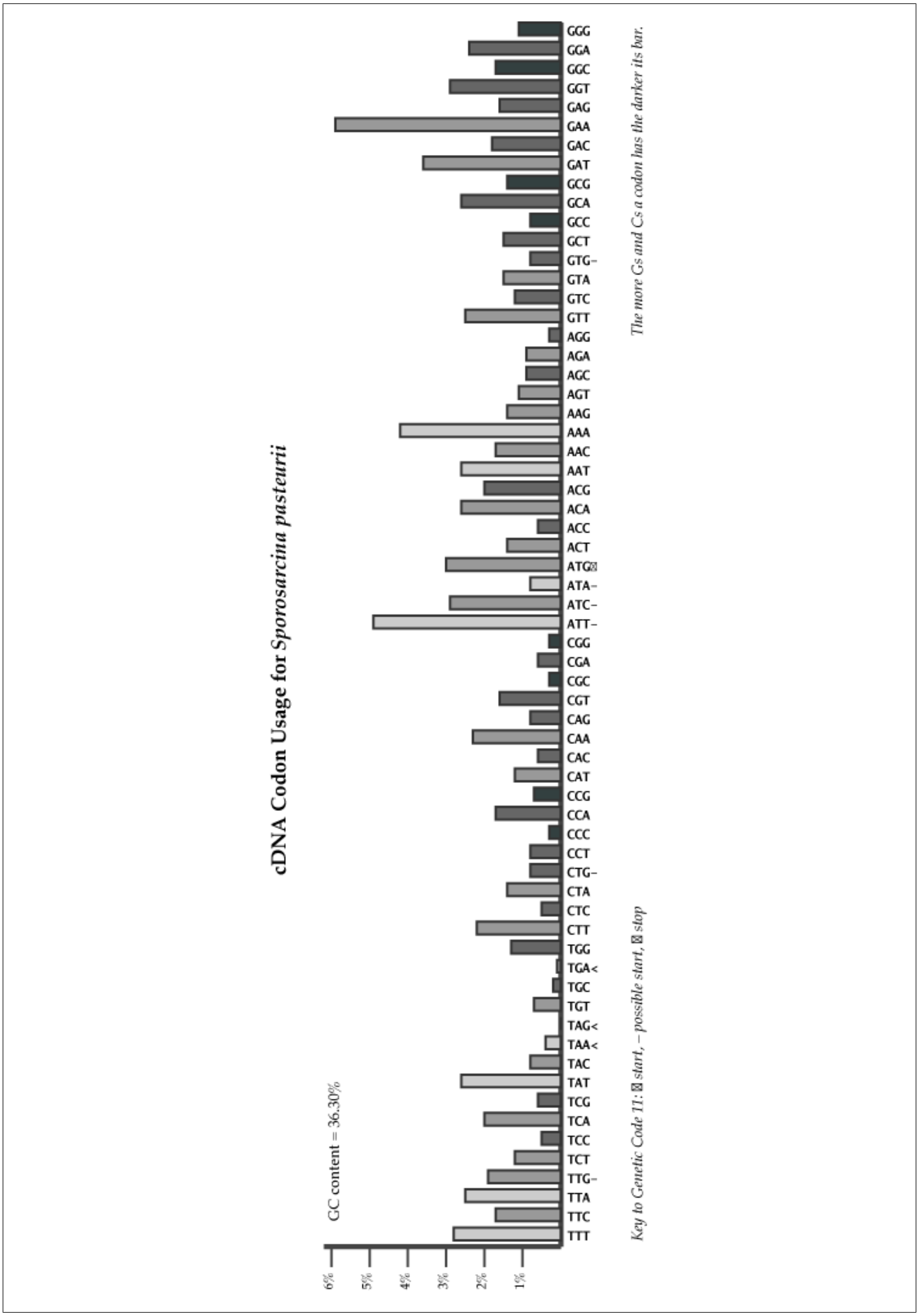


Figure 11-14. Chart of Sporosarcina pasteurii codon usage

The opening part of the SVG file generated by this example, containing the required invocations, is shown in Example 11-15. Next is a set of CSS style declarations. The details are omitted here, but the beginning and end are shown to indicate how to include CSS styles in an SVG document. (Example 11-15 includes a small section with actual CSS declarations.) The rest of the SVG file relies on these CSS declarations to describe things like color and font properties. A CSS declaration can specify properties in a variety of ways, including for a type of tag (e.g., `text`) or for all tags with a certain value of the `class` attribute. Finally, a few of the graphical element tags from each part of the file are included, with ellipses indicating more of the same.

Example 11-15. SVG for dispaying a codon usage bar graph

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns='http://www.w3.org/2000/svg' version='1.1'
     width='984' height='379'
>

  <defs>
    <style type='text/css'><![CDATA[
        ...CSS styles go here...
      ]]>
    </style>
  </defs>

  <!-- Key -->
    <text x='100' y='339' class='key' >
      Key to Genetic Code 11:
      &#10033; start, &#8211; possible start, &#8743; stop
    </text>
    <text class='keyright' x='944' y='310' >
      The more Gs and Cs a codon has the darker its bar. </text>
    <text x='100' y='72' >GC content = 36.30% </text>

  <!-- Title -->
    <text class='title' x='492' y='30' >
      cDNA Codon Usage for
      <tspan font-style='italic'
             text-anchor='middle' dominant-baseline='middle' >
        Sporosarcina pasteurii
      </tspan>
    </text>

  <!-- Axes -->
    <line class='axis' x1='80' y1='246' x2='944' y2='246' />
    <line class='axis' x1='80' y1='60' x2='80' y2='246' />
  <g class='tic'>
    <line x1='64' y1='216' x2='80' y2='216' />
    <text x='60' y='217' > 1% </text>
    <line x1='64' y1='186' x2='80' y2='186' />
```

```
    <text x='60' y='187' > 2% </text>
    . . .
  </g>

  <!-- Codon Labels -->
  <g class='codon'>
    <text x='191.5' y='251' transform='rotate(90,191.5,251)' >TAT</text>
    <text x='205.0' y='251' transform='rotate(90,205.0,251)' >TAC</text>
    <text x='218.5' y='251' transform='rotate(90,218.5,251)' >TAA&lt;</text>
    <text x='232.0' y='251' transform='rotate(90,232.0,251)' >TAG&lt;</text>
     . . .
  </g>

  <!-- Bars -->
    <rect class='gc0' x='83.5' y='162' width='10' height='84' />
    <rect class='gc1' x='97.0' y='195' width='10' height='51' />
    <rect class='gc0' x='110.5' y='171' width='10' height='75' />
    <rect class='gc1' x='124.0' y='189' width='10' height='57' />
    . . .

  </svg>
```

Sequencing trace file curves using SVG

Many curves in bioinformatics data presentations are highly irregular. This makes them difficult to draw using typical drawing facilities, such as those `tkinter` provides, or a description language such as SVG. A simplifying trick that often works for these kinds of curves is to draw a line from each point in the data to the next. The lines will be so small that the overall image will appear to be a curve.

A classic example is the display of trace data obtained from sequencing machines. One approach using SVG is simply to insert a `line` tag from every <x, y> position to the corresponding <x+1, y'> position. It turns out that the lines drawn are so small that the end result looks like a curve.

A file containing enough line tags for a sequencing trace will be quite large. Worse, a browser would get bogged down interpreting the many thousands of tags it includes. The SVG `polyline` is the perfect fix for this problem: using one dramatically shrinks the size of the file and, even more, the time required to process it, since instead of a complete line tag, with its various attributes, all that is needed for each point is an <x, y> pair. Example 11-16 shows some of the code that implements this approach. Figure 11-15, which follows the pages of code, shows the result.

> It might help you to understand the code in the two SVG examples if you pay more attention to its output and the way it looks in your browser than to the Python code. The Python code isn't all that intriguing: it just does what it needs to do. What's interesting is how a structured graphical image can be constructed from a structured text format.

Example 11-16. Drawing a sequence trace using SVG polylines

```
colors = ('black', 'green', 'red', 'blue')          # or whatever
axiscolor = '#333'
axisthickness = 2

tickcolor = '#666'
tickspacing = 100                                   # spacing of x-axis labels
tickheight = 18
tickthickness = 2

maxval = 1600
leftmargin = 10
topmargin = 20
yzero = 100

hscale = 5/2                                        # multiply x coordinates by this
vscale = 14                                         # divide y values by this
span = 1                                            # number of points to average together

def read_data(infilname):
    with open(infilname) as infil:
        return [eval(line) for line in infil.readlines()]

def write_text(fil, x, y, txt, cls):
    print("  <text class='{}' x='{}' y='{}'>{}</text>".
          format(cls, x+leftmargin, y+topmargin, txt),
          file=fil)

def write_line(fil, frompos, fromval, topos, toval, color, thickness):
        print("  <line x1='{}' y1='{}' x2='{}' y2='{}'".
              format(leftmargin + int(frompos * hscale),
                     int(((maxval - fromval) / vscale)) + topmargin,
                     leftmargin + int(topos * hscale),
                     int(((maxval - toval) / vscale)) + topmargin),
              end=' ',
              file=fil),
        print("style='stroke: {}; stroke-width: {};'/>".
              format(color, thickness),
              file=fil)

def write_axes(fil, count):
    print(file=fil)
    write_line(fil, 0, 0, 0, maxval, axiscolor, axisthickness)
    write_line(fil, 0, 0, count, 0, axiscolor, axisthickness)
    print(file=fil)
    tickbottom = vscale * (-axisthickness - tickheight)
    for x in range(int(tickspacing / hscale),
                   count,
                   int(tickspacing // hscale)):
        write_line(fil, x, tickbottom + vscale * (1 + tickheight), x, tickbottom,
                   tickcolor, tickthickness)
        write_text(fil,
                   int(x * hscale),
                   int((maxval / vscale)) + tickheight +1 5,
```

```
                  int(x * hscale),
                  'tick')

def write_bases(outfil, start, bases, positions):
    for base, pos in zip(bases, positions):
        write_text(outfil,
                   int(pos * hscale - 3),
                   int((maxval / vscale)) + 15,
                   base,        # text
                   base)        # class
    # - to "center" the base at its position; just a guess
    print(file=outfil)

def print_point(outfil, x, y):
    print("{},{}".format(leftmargin + int(x * hscale),
                         int((maxval - y) / vscale) + topmargin),
          file=outfil,
          end=' '
          )

def write_curve(outfil, vals, color):
    # at a point in the curve instead of using every data point
    # hscale is how far apart the x points are
    print("""
  <polyline style='fill: none; stroke: {}; stroke-width: 1;'
            points='""".
          format(color),
          file=outfil,
          end=''),

    # span is a file-level parameter that specifies the number of points to average to compute
    # a value rather than having a value for every point on the x axis; blur does the averaging
    for pos in range(0, len(vals)-1, span):
        if not pos % 6:                                  # 5-digit x's at the end
            print(20*' ', file=outfil, end='')
        print_point(outfil, pos/span, blur(vals[pos:pos+span]))

    print("  ' />", file=outfil)                         # close polyline

def write_curves(outfil, data,):
    for d, clr in zip(data, colors):
        write_curve(outfil, d, clr)

def blur(seq):
    """return the average of the values in seq"""
    return 0 if not seq else sum(seq)/len(seq)

def write_heading(outfil, width, height):
    print(
"""<?xml version="1.0" standalone="no"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
```

```
<svg xmlns='http://www.w3.org/2000/svg' version='1.1'
     width='{0}' height='{1}'
>""".format(width+(2*leftmargin), height+yzero+20),
"""
  <defs>
    <style type='text/css'><![CDATA[
        text {
              font-family: Futura, 'Andale Mono', Verdana, sans-serif;
              fill:black;
              font-weight: normal;
              font-size: 8pt;
              font-style: normal;
              text-rendering:optimizeLegibility;
            }
        text.tick { }
        text.T { fill: red; }
        text.C { fill: blue; }
        text.A { fill: green; }
        text.G { fill: black; }
      ]]>
    </style>
  </defs>
""",  file = outfil, end='\n')

def write_closing(outfil):
    print('\n</svg>', file = outfil)

def write_svg_file(infilname, outfilname):
    data = read_data(infilname)
    start = data[0]
    bases = data[1]
    positions = data[2]
    values = data[3:]
    with open(outfilname, 'w') as outfil:
        write_heading(outfil, len(values[0]) // span, maxval // vscale)
        write_axes(outfil, len(values[0]) // span)
        write_bases(outfil, start, bases, positions)
        write_curves(outfil, values)
        write_closing(outfil)

write_svg_file('data/ABI/seqdata', 'data/ABI/abi-trace.xml')
```

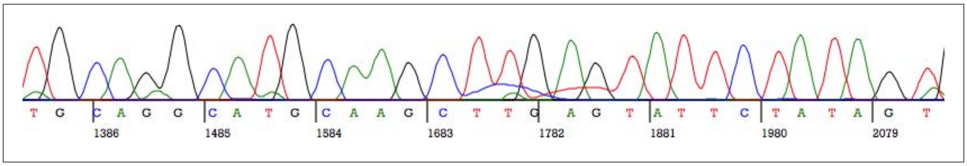


Figure 11-15. A portion of the generated SVG trace display

Tips, Traps, and Tracebacks

Tips

- The first time you use a graphics library such as `tkinter` or a language such as SVG, the first program you write should be something very minimal. Before starting any real work, make sure that you know how to create a diagram, add a shape, display the diagram, and close it.
- *Before doing any programming, sketch a high-level layout on paper*. Draw vertical and horizontal lines at each significant location in your sketch. Then, invent labels for the vertical and horizontal distances in each segment of the resulting grid. Use these labels in the code you write. Figure 11-4 shows an example of what a sketch like this layout might look like, but do it by hand.
- In any kind of programming it is best to avoid using any unnamed significant numbers, but in graphics programming it is essential. Graphics code is full of numbers and computations for positions, dimensions, scaling factors, font sizes, border widths, and so on. This is the "domain" of graphics programs. You cannot avoid the numbers, but you can avoid a great deal of effort, frustration, and errors by naming every value that goes into the computation of positions and dimensions, if for no other reason than that your names will all be self-documenting.

  If you decide to change the left margin from 12 to 18 but leave the right margin at 12, how will you know which 12s were for the right margin and which the left? Don't write code with a comment near the 12 that says "left margin." That's what names are for! In object-oriented programming class fields are the way to give these values names, as demonstrated in the programs in the first half of this chapter. For example, the histogram examples in this chapter even had a name for the number of pixels to use—3—for each 1/10 of a percent in the data. It's all right to add or subtract 1 or divide by 2 or 3 without giving the results names, but make sure the significance of those small values is trivial, not something that is controlling the nature of the graphics being constructed.
- As you develop the parts of the drawing, give each one a colored outline or background (perhaps even a different color for each one). Do this also for aspects of a region's border—that is, use different colors for padding, the border line itself, and its outside margin, depending on which of these are supported by the graphics software you are using. Coloring like this will help you detect positions and dimensions that are slightly "off" and line up pieces of the drawing.
- You don't have to develop the real parts of the drawing right away. Put in lines, rectangles, or circles that represent the positions and dimensions of the pieces you intend to add eventually. These are the graphics equivalent of `pass` statements.
- Concentrate on getting one part working at a time. Most programs are complex because of the interdependence of their components. Graphics programming adds

the further complication that the components of its *output* are similarly interdependent. In this chapter's examples we've seen titles, keys, axes, tics, tic labels, margins, and other elements that surround the content part of the graphics. The positions and dimensions of each of these may be affected by the positions and dimensions of several others. By programming just one component at a time you will not only avoid getting overwhelmed by the complexity of your program, but you will progressively develop a better understanding of the interrelationships among the graphical components.

- Programs that draw structured graphics can almost always be organized in several levels, as shown in this chapter's examples. High-level code simply organizes and sequences the construction of the diagram. Lower-level code typically just draws a single line or piece of text, etc. The real work is done in the middle-layer code that is called from the high-level code. The purpose of the lower-level code is to extract the repetitive patterns of method calls and parameters in the middle-level code. This can simplify the middle-level code considerably. In addition, as we saw with the `tkinter`-based classes in this chapter's examples, organizing your code like this will reveal similarities among the various graphics programs you write.
- You will often be able to derive a new program from an existing one with relatively few changes. Once you start doing that, you can capture the generalities in an abstract superclass and make subclasses that define only the differences from that class. The first few subclasses you develop will lead to many changes in all the classes, but after a few the superclass will generally stabilize, and you'll be able to use it easily.
- It is more important in graphics programming than in almost any other kind of software development to keep multiple versions of your files. You don't need anything fancy, just a way to save a file with a numerical suffix in a different directory. Every time you make a significant change, you should save the file both to its normal location and with an incremented backup suffix. For example, if you have five backup versions of *diagram.py* and have just reworked the code to take into account the border widths of some components, save it both to *diagram.py* and to *save/diagram06.py*. (Use the zero in version numbers less than 10 so that file listings are sorted appropriately.)

Traps

- "Off by one" errors and issues are pervasive in graphics programming. Avoiding them is partly a matter of learning how the tools you are using behave. For instance, a horizontal line of width 3 from (0, 10) to (50, 10) may cover the y coordinates 7–9, 8–10, 9–11, 10–12, or 11–13, depending on the software used and perhaps on parameters specified when the line is drawn. Much of the problem, though, is caused by the interdependence of component positions and dimensions. Graphics code contains a lot of computations of positions and dimensions where each

value is a combination of several others, and often you will need to add 1 to or subtract 1 from some of these values to get things lined up properly.

- The same concerns apply to the borders of rectangles, along with some additional complications. Do their width and height values include the padding? The border? Half of the border? You'll need to become familiar with the peculiarities of your software.
- Intricacy is unavoidable in graphics code. It is rarely possible to achieve the kind of elegant terseness one should strive for in most other kinds of programming tasks. You will see just by looking at graphics code that it has an entirely different "shape" or "feel" than other code you have written or read.

SVG traps

- As in HTML, a sequence of one or more whitespace characters is treated as a single space.
- The value of each attribute must be quoted.
- Attributes are separated by spaces, not commas, semicolons, or any other punctuation.
- Style properties are `*name*:*value*` pairs separated by semicolons; whitespace may appear after the colon.
- All but text tags must be closed with a slash (i.e., `/>`); text tags are closed with `>`, then followed by text and a closing `</text>`.
- Font properties, stroke, stroke width, fill, etc., are *style* properties, not *tag* attributes; they must be specified as the value of a tag's `style` attribute or in the stylesheet for the document.
- SVG's rotation operation is confusing: be sure to specify the second and third arguments for rotate transformations, or the results will almost definitely not be what you expect. In fact, rotating without an origin will usually make the element disappear from the diagram. In general, you'll want to rotate elements around either their starting points or their centers.
- CSS has a concept of "class," which is used to specify style details for all tags with a particular value for the `class` attribute. However, remember that `class` is a keyword in Python, used to define classes. Don't try to use it for a variable in code that deals with CSS classes; call it `cls` or `cl` or something similar instead. It would be nice if syntax errors caused by the use of a keyword were flagged as such in the Python error message, rather than the message simply pointing to the keyword, but that's not the case.

Tracebacks

Here's an error message you might encounter programming with `tkinter`:

`_tkinter.TclError: unknown color name "grue"`
: A typical, and fairly helpful, `tkinter` error message.

> One surprising thing about this error message is the underscore that precedes `tkinter` in its name. You will sometimes see single underscores at the beginning of names of modules that are interfaces to external libraries, such as Tk and SQLite3, or Python libraries that are implemented in C for one reason or another. They occasionally appear at the beginning of class names too. Don't worry about them—they are just ordinary names. (You can't ignore the underscores if you are trying to catch an exception whose class or module name begins with one, though: you have to give an `except` clause the exact name of the error class.)

APPENDIX A

Python Language Summary

This appendix summarizes the Python language as used in this book. Some of its more obscure features have been omitted.

Language Components

Special Syntactic Elements

`#`
: Begins a comment; the rest of the line is ignored

`\`
: At end of line, indicates that line break should be ignored

`''` *or* `""`
: Enclose a one-line string

`''' '''` *or* `""" """`
: Enclose a multiline string

`()`
: Enclose an expression or call a function

Keywords

Table A-1 shows the complete list of Python's keywords. Note that a few of them play multiple roles:

- `not` is a logical operator and part of the `not in` and `is not` operators.
- `in` is an operator as well as part of a `for` statement.
- `as` names things in both `with` statements and `except` clauses.

Table A-1. Keywords

Kind	Keywords	Note
Value names	`None True False`	
Operators	`not or and in`	It is the *words* that are keywords: `not` is both a Boolean operator and part of the `not in` and `is not` operators
	`is`	
	`del`	
General	`pass assert`	
	`return yield`	
Definitions	`def class as`	
	`import from`	
	`global nonlocal`	These two keywords affect the scope of assigned names; they are not used in any of the book's code examples.
	`lambda`	
Conditionals	`if else elif`	
Loops and iterations	`while for break continue`	Also `in`, included with the operators, as part of a `for` statement
Exception handling	`try except finally`	`with` is included here because it creates an implicit `try`/`finally` exception handler
	`with raise`	

Special Names

In addition to keywords, the names described in Table A-2 have special significance. An asterisk stands for any number of characters that can appear in a name.

Table A-2. Names with special significance

Names	Significance
`_`	In the interpreter only, refers to the result of the last evaluation
`_*`	Used for names not imported by `from module import *`
`__*`	Used for names private to their classes
`__*__`	Reserved for system-defined method names used for implementation of fundamental operations

Operators

Boolean operators are listed in Table A-3.

Table A-3. Boolean operators

Operator	Meaning
`not`	Unary: `True` if operand is `False`; `False` if operand is `True`
`or`	`True` if either operand is `True`; otherwise, `False`
`and`	`True` if both operands are `True`; otherwise, `False`

Table A-4 lists Python's arithmetic operators. Two of them can be used with sequences:

- `+` produces a new sequence with all the elements of each of its sequence operands.
- `*` takes a sequence and an integer and produces a new sequence with that number of repetitions of the original.

Otherwise, all operands must be numbers. If any of the operands is a `float`, the result will be a `float`.

Table A-4. Arithmetic operators

Operator	Meaning
`+`	Unary positive, binary addition, sequence concatenation
`-`	Unary negative, binary subtraction
`*`	Number multiplication, sequence repetition
`**`	Exponentiation
`/`	Division; *result is always a float*
`//`	Floor division; result *equals* an integer, but its *type* is an integer only if both arguments are integers
`%`	Remainder (modulo) of dividing the first operand by the second

Floor division and modulo are related values that are returned as tuples by the built-in function `divmod`:

```
divmod(n, m) == (n//m, n%m)
n == ((n // m) * m) + (n % m)
```

Python's comparison operators are shown in Table A-5. The operands of equality and inequality may have different types, but two values of different types are always unequal. In general, it is an error to try to compare values of different types using one of the order comparisons.

Table A-5. Comparison operators

Operator	Meaning
`<`	op1 is less than op2
`>`	op1 is greater than op2
`<=`	op1 is less than or equal to op2
`>=`	op1 is greater than or equal to op2
`==`	op1 is equal to op2
`!=`	op1 is not equal to op2
`is`	op1 is *identical* to op2 (i.e., they are the same object)
`is not`	op1 is *not* identical to op2
`in`	op1 is in the collection op2
`not in`	op1 is not in the collection op2

Table A-6 lists the operators that perform bitwise operations. Their operands must be integers.

Table A-6. Bitwise operators

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td><code><<</code></td><td>Shift op1 left by op2 bits</td></tr>
<tr><td><code>>></code></td><td>Shift op1 right by op2 bits</td></tr>
<tr><td><code>&</code></td><td>Each bit is the and of the corresponding bit of the two operands</td></tr>
<tr><td><code>|</code></td><td>Each bit is the inclusive or of the corresponding bit of the two operands</td></tr>
<tr><td><code>^</code></td><td>Each bit is the exclusive or of the corresponding bit of the two operands</td></tr>
<tr><td><code>~</code></td><td>A unary operator: each bit is the complement of the corresponding bit in the operand</td></tr>
</table>

Anonymous Functions

A lambda expression creates an anonymous function:

```
lambda x, y: expression using x and y
```

A lambda takes one or more arguments. Only one expression—not statement—is allowed.

Types and Expressions

Table A-7 lists Python's primitive types. Collection types are summarized in Appendix B.

The `str` type may be considered a primitive for some purposes, and a collection for others.

Table A-7. Primitive types

Type	Description
`None`	Special "no-value" value
`bool`	`True` or `False`
`int`	Integers (any magnitude)
`float`	"Floating-point" numbers
`str`	Unicode strings

Table A-8 shows the syntax for constructing expressions. (Sequences, of which strings are an example, are fully documented in Appendix B.)

Table A-8. Expressions

Kind of operator	Syntax	Notes
Numeric	`+ - * / // % **`	/ is `float` division; // is `int` division; % is modulo; and ** is power
Logical	`not and or`	
Comparison	`== != < <= >= >`	
Sequence	`+, in, not in`	
Sequence subscription	`[]`	
Sequence slicing	`[m:n], [m:n:k]`	Omitting `m` is equivalent to `0`, omitting `n` is equivalent to the end of the sequence
Function call	`fn(), fn(arg1,...)`	
Method call	`arg1.fn(arg2,...)`	

Statements

Table A-9 describes all the simple statements in Python. (The `del` statement is included here for completeness; it applies only to collection types, which are described in Appendix B.)

Table A-9. Simple statements

Statement	Description
name = *`expression`*	Binds *name* to the value of *`expression`*
name1 = *name2* = `...` = *`expression`*	Binds the names to the value of *`expression`*
`pass`	Do-nothing placeholder used where statements are required, e.g., in a function definition
`return`	Returns None from a function
`return` *`expression`*	Returns value of *`expression`* from a function
`yield`	Makes the function containing it a generator, with every execution of `yield` producing one value
`assert` *`expression`*	Raises an error if the expression is false
`assert` *`expression`*, *`value`*	Raises an error if the expression is false, including *`value`* in the error message
`import` *`module`*	Adds the contents of *`module`* to Python and adds its name to the namespace in which the statement is executed
`from` *`module`* **`import`** *name*	Adds the contents of *`module`* to Python and adds *name* to the namespace in which the statement is executed
`from` *`module`* **`import`** *name* `as` *`alias`*	Adds the contents of *`module`* to Python and adds *`alias`* to the namespace in which the statement is executed, with *`alias`* used instead of *name*
`del` *`coll`*`[...]`	Removes the element(s) from *`coll`* (a list or dictionary) indicated by an index or subscription

Statement	Description
continue	Interrupts the current loop or iteration
break	Exits the current loop or iteration
raise ExceptionClass(*arg*)	Creates an instance of ExceptionClass and raises an exception
global *name1*, *name2*, ...	Must be first statement of a function; indicates that assignments of the listed names will be in the global scope
nonlocal *name1*, *name2*, ...	Must be first statement of a function; indicates that assignments of the listed names will be in the immediately enclosing scope; used when one function is defined inside of another

Table A-10 lists Python's augmented assignment statements. Both ***name*** and ***expression*** must have numeric values. (The **+=** operator also works when the value of ***name*** is a list, in which case the value of ***expression*** may have any type; see Appendix B.) The statements on the lefthand side of the table rebind ***name*** to the new value, with the same effects as the statement on the righthand side of the same row of the table.

Table A-10. Augmented numeric assignment statements

Statement	Equivalent
name += *expression*	*name* = *name* + *expression*
name -= *expression*	*name* = *name* - *expression*
name *= *expression*	*name* = *name* * *expression*
name /= *expression*	*name* = *name* / *expression*
name //= *expression*	*name* = *name* // *expression*
name %= *expression*	*name* = *name* % *expression*
name **= *expression*	*name* = *name* * *expression*

There are a very small number of compound statements in Python that define new names. They are shown in Table A-11.

Table A-11. Compound statements that define names

Statement	Description
def *name*(*parameter-list*): *statements*	Binds name to a new function whose definition is specified by *parameter-list* and *body*, which consists of one or more statements indented relative to the word def.
class *Name*: *statements*	Defines a class, with *statements* one or more statements indented relative to the word class; the statements are usually defs, occasionally assignments, and very rarely anything else.
class *Name*(*SuperName*): *statements*	Defines a class with *SuperName* as its superclass.
with *expression* **as** *name*: *statements*	1. Evaluates the expression. 2. Binds *name* to the result.

Statement	Description
	3. Executes the *`statements`*. 4. Whether an error occurs or not, performs special actions to undo the effect of evaluating the expression. Used most frequently with *`expression`*, a call to open.

Table A-12 describes Python's conditional statements.

Table A-12. Conditional statements

Statement	Description
`if` *`expression`*`:` *`statements`*	Evaluates *`expression`* and, if true, executes the statements

`if` *`expression`*`:`
    `statements1`
`else:`
    `statements2`
	Evaluates *`expression`* and, if true, executes *`statements1`*; otherwise, execute *`statements2`*

`if` *`expression1`*`:`
    `statements1`
`elif` *`expression2`*
    `statements2`
`. . .`
`else:`
    `default-statements`
	Evaluates each expression in sequence until one is true, then executes the corresponding statements; if none is true, executes *`default-statements`*

Table A-13 describes Python's loop and iteration statements. See Appendix B for summaries of common uses with collections.

Table A-13. Loop and iteration statements

Statement	Description
`while` *`expression`*`:` *`statements`*	Repeatedly evaluates the *`expression`* and executes the *`statements`* until the *`expression`*'s value is false

`while` *`expression`*`:`
    `statements1`
`else:`
    `final-statements`
	Repeatedly evaluates the *`expression`* and executes the *`statements`* until the *`expression`*'s value is false; then evaluates *`final-statements`*

`for` *`item`* `in` *`collection`*`:` *`statements`*	For each item in *`collection`*, executes the *`statements`*

An exception statement begins with the keyword `try` and must have at least one of the other kinds of clauses. There can be any number of `except` clauses that list one or more exception classes. There may not be more than one `except` clause that does not name an exception class, nor more than one `finally` clause.

When an exception is raised during the execution of the `try` block (that is not handled by a "deeper" `try` statement), that block is immediately exited. Next, the type of the exception raised is compared to the type(s) of each `except` clause until a match is found, at which point the statements of that clause are executed and execution continues after

the last except clause. If an except clause without any classes is reached, its statements are executed.

Note that it is possible for all the except clauses to list one or more class names, none of which match the type of the exception raised. In that case, the try statement does not catch the exception, and Python continues looking up the stack for a statement that does catch the exception.

Regardless of whether the try statements completed without an exception being raised or were interrupted by an exception, if there is a finally clause, its statements will be executed.

The components of an exception handling statement are shown in Table A-14. Note that what appears in the lefthand side of the statement are *clauses*, not *statements*. A try statement *must* try and have one or more except clauses. It *may* have one finally clause.

Table A-14. Exception handling statements

Statement	Description
try: *try-statements*	Executes the *try-statements*
except *ExceptionClass*[, ...]: *except-statements*	If an exception that is an instance of one of the listed types is raised, executes the *except-statements*; then continues after the last except clause of the try statement
except *ExceptionClass*[, ...] **as** *name*: *except-statements*	As in the form described in the previous row, but also names the exception object so it can be referenced in the *except-statements*
finally: *finally-statements*	Executes the *finally-statements* regardless of whether an error occurred or not; must be the last in the series of clauses that follow the *try-statements*

Notes

Two features of Python syntax were omitted from this book because they disturb the regularity of code appearance. Since you may see these in other code you read, though, you should know about them:

1. Strings can be concatenated during compilation, as opposed to execution, simply by putting them next to each other with no separating +. They can even appear on separate lines, as long as all that is between them is whitespace. For example:

   ```
   print('This string might be too long to fit on one line where'
         ' it appears in your program, so just write two or more'
         ' strings in succession')
   ```

   There's no particular advantage to doing this rather than using a +, except in the rare case where a statement containing a concatenation of string literals (not names whose values are strings) would otherwise be executed a very large number of

times. Concatenation at compile time creates a single string. Concatenation at runtime creates a new string every time a string concatenation expression (*str1* + *str2*) is evaluated.

2. When the body of a one-clause compound statement (or the body of a clause of a multiclause compound statement) consists of a single statement, it can appear on the same line as the first line of the clause, after the colon. For example:

```
if value: print(value)
else: print('None found')
```

APPENDIX B
Collection Type Summary

This appendix provides a compact summary of (most of) Python's collection types. It describes the characteristics and methods of each collection and the operators and functions used with collection values. At the end of the appendix, you will find summaries of the collection iteration templates shown in Chapter 4.

Types and General Operations

Table B-1 summarizes the characteristics of Python's most commonly used collection types. The column labeled "Syntax" shows examples of two-element collections, with *`e1`* and *`e2`* representing elements of any type and *`k1`* and *`k2`* representing instances of an immutable type.

Table B-1. Collection types

Element access	Name	Type	Element type	Mutable?	Syntax
Unordered unique elements	Set	`set`	Any immutable	Yes	`{`*`k1`*`, `*`k2`*`}`
	Frozenset	`frozenset`	Any immutable	*No*	*None*
Ordered (indexed)	String	`str`	One-character strings[a]	*No*	`'ac'` `"ac"` `'''ac'''` `"""ac"""`
	Bytes	`bytes`	8-bit bytes[b]	*No*	`b'ac'` `b"ac"` `b'''ac'''` `b"""ac"""`
	Bytearray	`bytearray`	8-bit bytes	Yes	*None*
	Range	`range`	Integers	*No*	*None*
	Tuple	`tuple`	Any	*No*	`(`*`e1`*`, `*`e2`*`)`
	List	`list`	Any	Yes	`[`*`e1`*`, `*`e2`*`]`
Associative	Dictionary	`dict`	Keys immutable, values any	Yes	`{`*`k1`*`:`*`e1`*`,`

Element access	Name	Type	Element type	Mutable?	Syntax
					k2:*e2*}
Stream (next)	File object	Depends[c]	Characters, bytes, lines	Depends on use	*None*

[a] Strings don't really "contain" characters or one-character strings, but for many purposes they behave as if they do.

[b] Bytes are equivalent to the integers from 0 through 255.

[c] The type of the object returned by open is an instance of the class from the IO library; it isn't one of the built-in collection types.

Common Operations

For all sequence types except `range`, calling the type with no arguments produces an empty sequence, and calling the type with a sequence argument produces a new object of the type containing the elements of the argument.

The two byte types—`bytes` and `bytearray`—can be called with an integer to create a sequence of that many zero bytes.

When `dict` is called with a sequence argument the elements of the sequence must all be sequences of length 2 containing a key and a value, in that order.

Except for `range`, all sequence types implement the methods `count` and `index`. Table B-2 lists the operations and functions that apply to *all* collection types.

Table B-2. Operations and functions common to all collection types

Function/operation	Returns
`x in coll`	`True` if *coll* contains x.
`x not in coll`	`True` if *coll* does not contain *x*.
`any(coll)`	`True` if any item in *coll* is true; otherwise, `False`.
`all(coll)`	`True` if every item in *coll* is true; otherwise, `False`.
`len(coll)`	The number of items in *coll*.
`max(coll)`	The maximum item in *coll*, which may not be empty.
`min(coll)`	The minimum item in *coll*, which may not be empty.
`sorted(coll[, keyfn] [, reverseflag])`	A list containing the elements of *coll*, sorted by comparing elements or, if *keyfn* is included in the call, comparing the results of calling *keyfn* for each element; if *reverseflag* is true, the ordering is reversed. *keyfn* and *reverseflg* must be specified as keyword arguments, not positionally.

Creating Collections

As with all types, collection types can be called to create new instances. Table B-3 shows the details of the parameters of each type's call.

In the Python documentation, you will find these functions in the section "Built-in Functions," under "Built-in Types."

Table B-3. Collection creation calls

Method	Description
`set([`*iterable*`])`	
`frozenset([`*iterable*`])`	
`str([`*source*`[, `*encoding*`]])`	If no *source*, returns an empty string; if no *encoding*, returns the result of *source*'s `__str__` method if it has one, or `__repr__` if not. If the type of *source* is `bytes` or `bytearray`, decodes it using *encoding*.
`bytes([`*source*`[, `*encoding*`]])`	If *source* is an integer, returns a `bytes` instance of that many null bytes; an iterable of integers `0<=x<=255` is used to initialize the new `bytearray`. If *source* is a string, *encoding* must be specified (e.g., `'utf8'`) and will be used to encode *source*.
`bytearray([`*source*`[, `*encoding*`]])`	Same as with `bytes`.
`range([`*start*`], `*stop*`[, `*step*`])`	Arguments must be integers; returns a `range` object from *start* (default 0) to *stop* in increments of *step* (default 1).
`tuple([`*iterable*`])`	Returns a tuple with the items of *iterable* in the order in which they appear there.
`list([`*iterable*`])`	Returns a list with the items of *iterable* in the order in which they appear there.
`dict([`*arg*`])`	With no *arg*, returns an empty dictionary; with *arg* a dictionary, returns a copy. Otherwise, *arg* must be an iterable of key/value pairs.
`open(`*filename*`, `*mode*`)`	Creates a file object.

Another way to create collections is with comprehensions:

```
List comprehension:
[expression for item in collection]

Set comprehension:
{expression for item in collection}

Dictionary comprehension:
{key-expression: value-expression
 for key, value in collection}
```

The collection in a dictionary comprehension must consist of two-element tuples or lists.

Conditions can be added to comprehensions. There can be any number of tests, and all have to be true for the element to be included in the result. Here are some examples for lists:

```
[expression for item in collection if test]
[expression for item in collection if test1 if test2]
```

Comprehensions can have more than one `for`. For example:

```
[c1 + c2 + c3 for c1 in 'TCAG' for c2 in 'TCAG' for c3 in 'TCAG']
```

Specific Collection Types

This section summarizes the nature of the collection types used in this book and the functions, methods, and expressions that operate on them.

Sets

A *set* is an *unordered* collection of items that contains *no duplicates*.

Table B-4 lists comparison operations that can be performed on sets. Some are expressed using an operator whose operands are both sets, some with a method call whose argument is any kind of collection, and some with either.

Table B-4. Set comparison operations

Operator	Method	Result
	set1.isdisjoint(*coll*)	True if the set and the argument have no elements in common
set1 <= *set2*	*set1*.issubset(*coll*)	True if every element of *set1* is also in *set2* (*coll* for the method)
set1 < *set2*		True if every element of *set1* is also in *set2* (*coll* for the method) and *set2* is larger than *set1* (i.e., *set1* is a proper subset of *set2*)
set1 >= *set2*	*set1*.issuperset(*coll*)	True if every element of *set2* (*coll* for the method) is also in *set1*
set1 > *set2*		True if every element of *set2* (*coll* for the method) is also in *set1* and *set1* is larger than *set2* (i.e., *set2* is a proper subset of *set1*)

Table B-5 lists operations that produce new sets from the elements of other sets or collections. Each can be expressed using an operator whose operands are both sets. Three of them have an equivalent method whose arguments are any number of any kind of collection. One has an equivalent method that takes just one collection of any type.

Table B-5. Algebraic set operations

Operator	Method	Result
set1 | *set2*	*set1*.union(*coll*, ...)	A new set with the elements of both the set and the arguments
set1 & *set2*	*set1*.intersection(*coll*, ...)	A new set with the elements that are common to the set and the arguments

Operator	Method	Result
set1 - *set2*	*set1*.difference(*coll*, ...)	A new set with the elements that are in the set but not in the arguments
set1 ^ *set2*	*set1*.symmetric_difference(*coll*)	A new set with the elements that are in either the set or the arguments but not both

Table B-6 lists the operations and methods that actually modify the first set rather than returning a new set.

Table B-6. Set modification operations

Operator	Method	Result
set1 |= *set2*	*set1*.update(*coll*)	Updates *set1* by adding the elements in *set2* (*coll* for the method)
set1 &= *set2*	*set1*.intersection_update(*coll*)	Updates *set1* to keep only the elements that are in both *set1* and *set2* (*coll* for the method)
set1 - *set2*	*set1*.difference_update(*coll*)	Updates *set1* to keep only the elements that are in *set1* but not in *set2* (*coll* for the method)
set1 ^ *set2*	*set1*.symmetric_difference_update(*coll*)	Updates *set1* to keep only the elements that are in either *set1* or *set2* (*coll* for the method)
	set1.add(*item*)	Adds *item* to *set1*
	set1.remove(*item*)	Removes *item* from *set1*; it is an error if *item* is not in the set
	set1.discard(*item*)	Removes *item* from *set1* if it is present; no error if it is not present

Sequences

Sequences are *ordered* collections that *may contain duplicate elements*.

Table B-7 summarizes sequence indexing and slicing operations.

Table B-7. Sequence indexing and slicing

Type of seq	Indexing	Slicing	Slice assignment
	seq[*n*]	*seq*[*i*:*j*]	*seq*[*i*:*j*] = *obj*
str	Yes	Yes	*No*
bytes	Yes	Yes	bytes *no*
bytearray			bytearray yes
range	Yes	*No*	*No*
tuple	Yes	Yes	*No*
list	Yes	Yes	Yes

[Table B-8](#) summarizes other sequencing operations.

Table B-8. Sequence operators

Type of seq	Membership	Combination	Repetition	Comparison*
	obj in *seq*	*seq1* + *seq2*	*seq* * *n*	*seq* · *other*
	obj not in *seq*	(*seq1* and *seq2* have the same type)	(*n* an int)	with · one of ==, !=, >, <, >=, or <=
str	*obj* is a str	str	Yes	*other* is a str
bytes bytearray	*obj* is 0..255, a bytes, or a bytearray	Yes	Yes	*other* is a bytes or bytearray
range	*obj* is an int[a]	*No*	*No*	*No*
tuple	*obj* has any type	Yes	Yes	*other* is a tuple
list	*obj* has any type	Yes	Yes	*other* is a list

[a] Actually, the expressions *obj* in *rng* and *obj* not in *rng* are always valid for any type of *obj* and any range *rng*, but always False if *obj* is not an int.

[Table B-9](#) documents the way various forms of slice expressions work.

Table B-9. Slice expressions

Operation	Returns
seq[*i*:*j*]	Elements of *seq* from *i* up to, but not including, *j*
seq[*i*:]	Elements of *seq* from *i* through the end of the sequence
seq[:*j*]	Elements of *seq* from first up to, but not including, *j*
seq[:-1]	Elements of *seq* from first up to, but not including, the last
seq[:]	All the elements of *seq*—i.e., a copy of *seq*
seq[*i*:*j*:*k*]	Every k^{th} element of *seq*, from *i* up to, but not including, *j*

Strings

Strings are sequences of Unicode characters (though there is no "character" type). The `bytes` and `bytearray` types are sequences of single bytes. They have essentially the same operations and methods as `str`. One important difference is that with a string, regardless of whether you index an element or specify a slice, you always get back a string. With `bytes` and `bytearray`s, although slices return the same type, indexing an element returns an integer from `0` through `255`.

Since strings and bytes are immutable, no `str` or `byte` method modifies the string through which it is called. However, `bytearray` is a mutable type; elements of its instances may be replaced by using assignment, including assignment of slices.

[Table B-10](#) lists string predicate methods. All predicates return `False` for an empty string. For the "case" tests, the string must contain at least one character of the

appropriate type. Note that not all characters are "cased": digits and punctuation, for example, are not.

Table B-10. String predicates for kinds of characters

Method	Result
`isalpha`	`True` if all characters are alphabetic
`isalnum`	`True` if all characters are alphanumeric
`isdigit`	`True` if all characters are digits
`isnumeric`	`True` if all characters are numeric, including Unicode number characters
`isdecimal`	`True` if all characters are valid for use in decimal numbers
`islower`	`True` if all cased characters are lowercase
`isupper`	`True` if all cased characters are uppercase
`istitle`	`True` if all uppercase characters follow uncased characters and all lowercase characters follow cased characters

Table B-11 describes the string methods that search for one string inside another. In all the methods documented, ***`startpos`*** defaults to `0` and ***`endpos`*** defaults to the end of the string (inclusive)—i.e., ***`str2`*** is compared to ***`str1`***, ***`str1`***`[`***`startpos`***`:]`, ***`str1`***`[:`***`endpos`***`]`, or ***`str1`***`[`***`startpos`***`:`***`endpos`***`]`, according to which of the optional arguments are specified in a call.

Table B-11. String predicate and search methods

Method	Result
`str1``.startswith(`*`str2`*`[, `*`startpos`*`[, `*`endpos`*`]])`	`True` if *`str1`* starts with *`str2`*, taking into account *`startpos`* and *`endpos`*
`str1``.endswith(`*`str2`*`[, `*`startpos`*`[, `*`endpos`*`]])`	`True` if *`str1`* ends with *`str2`*, taking into account *`startpos`* and *`endpos`*
`str1``.find(`*`str2`*`[, `*`startpos`*`[, `*`endpos`*`]])`	Lowest index of *`str2`* in *`str1`*, taking into account *`startpos`* and *`endpos`*; `-1` if not found
`str1``.rfind(`*`str2`*`[, `*`startpos`*`[, `*`endpos`*`]])`	Highest index of *`str2`* in *`str1`*, taking into account *`startpos`* and *`endpos`*; `-1` if not found
`str1``.index(`*`str2`*`[, `*`startpos`*`[, `*`endpos`*`]])`	Lowest index of *`str2`* in *`str1`*, taking into account *`startpos`* and *`endpos`*; error if not found
`str1``.rindex(`*`str2`*`[, `*`startpos`*`[, `*`endpos`*`]])`	Highest index of *`str2`* in *`str1`*, taking into account *`startpos`* and *`endpos`*; error if not found
`str1``.count(`*`str2`*`[, `*`startpos`*`[, `*`endpos`*`]])`	The number of occurrences of *`str2`* in *`str1`*, taking into account *`startpos`* and *`endpos`*

Table B-12 lists methods that return a new string with characters adjusted as necessary to the corresponding case.

Table B-12. String methods for changing case

Method	Result
`str1.lower()`	A copy of *str1* with all of its characters converted to lowercase
`str1.upper()`	A copy of *str1* with all of its characters converted to uppercase
`str1.capitalize()`	A copy of *str1* with only its first character capitalized; has no effect if the first character is not a letter (e.g., if it is a space)
`str1.title()`	A copy of *str1* with each word beginning with an uppercase character and the rest lowercase
`str1.swapcase()`	A copy of *str1* with lowercase characters made uppercase and vice versa

Methods that remove spaces from or add them to strings are described in Table B-13.

Table B-13. Methods for reformatting strings

Method	Result
`str1.lstrip([chars])`	A copy of *str1* with all characters until the first character not contained in *chars* (default is whitespace) removed
`str1.rstrip([chars])`	Like `lstrip`, but removes characters from the end
`str1.strip([chars])`	Like `lstrip`, but removes characters from both the start and the end
`str1.ljust(width[, fillchar])`	A string that is at least *width* long, padded with *fillchar* on the left (default is whitespace)
`str1.rjust(width[, fillchar])`	Like `ljust`, but the padding is on the right
`str1.center(width[, fillchar])`	Like `ljust`, but the padding is divided between the left and right
`str1.center([tabsize])`	A string with each tab of *str1* replaced with enough spaces to reach the next multiple of *tabsize* (default is 8)

Methods for joining and splitting strings are documented in Table B-14.

Table B-14. Methods for joining and splitting strings

Method	Result
`sepr.join(seq)`	A string formed by concatenating the strings in the sequence *seq*, separated by *sepr*, which can be any string (including the empty string)
`str1.splitlines([keepflg])`	A list of the lines in *str1*; if *keepflg* is `True`, the string includes end-of-line characters
`str1.split([sepr [, count]])`	A list of the "words" in *str1*, using *sepr* as a word delimiter (default is whitespace); if *count* is included as an argument, the result is limited to *count* words

Method	Result
str1.rsplit([*sepr* [, *count*]])	Like *split*, but starts from the end
str1.partition(*sepr*)	A tuple of three elements: the portion of *str1* up to the first occurrence of *sepr*, *sepr*, and the portion of *str1* after the first occurrence of *sepr*; if *sepr* is not found, the result is (*s*,'','')
str1.rpartition(*sepr*)	Like partition, but starts from the end

Table B-15 lists some general-purpose string replacement methods.

Table B-15. General-purpose string replacement methods

Method	Result
str1.replace(*oldstr*, *newstr*[, *count*])	Returns a copy of *str1* with all occurrences of the substring *oldstr* replaced by the string *newstr*; if *count* is specified, only the first *count* occurrences are replaced.
str1.translate(*dictionary*)	With *dictionary* having integers as keys, returns a copy of *str1* with any character *char* for which ord(*char*) is a key in *dictionary* replaced by the corresponding value; exactly what the replacement does depends on the type of the value in the dictionary, as follows: • None—character is removed from *str1* • Integer *n*—character is replaced by chr(*n*) • String *str2*—character is replaced by *str2*, which may be of any length
str.maketrans(*x*[, *y*[, *z*]])	(Called directly through the str type, not an individual string.) Produces a translation table for use with translate more conveniently than manually constructing the table; arguments are interpreted differently depending on how many there are: • *x*—character is removed from *str1* • *x*, *y*—*x* is a dictionary like that expected by translate, except that its keys may be either integers or one-character strings • *x*, *y*, *z*—*x* and *y* are strings of equal length; the table will translate each character of *x* to the character in the corresponding position of *y*

Lists

Lists can be modified through assignment statements, as shown in Table B-16.

Table B-16. List-modifying assignments

Statement	Result
lst[*n*] = *x*	Replaces the nth element of *lst* with *x*
lst[*i*:*j*] = *coll*	Replaces the ith through j^{th} elements of *lst* with the elements of *coll*

Statement	Result
lst[*i*:*j*] = *empty_coll*	Deletes the *i*th through *j*th elements of *lst* (an important special case of *lst*[*i*:*j*] = *seq*)
lst[*i*:*j*:*k*] = *coll*	Replaces the elements of *lst* designated by the slice with the elements of *coll*, whose length must equal the number of elements designated by the slice
lst[*n*:*n*] = *coll*	Inserts the elements of *coll* before the *n*th element of *lst*
lst[len(*lst*):len(*lst*)] = [*x*]	Adds *x* to the end of *lst*
lst[len(*lst*):len(*lst*)] = *coll*	Adds the elements of *coll* at the end of *lst*
lst += *coll*	
lst[:] = *coll*	Replaces the entire contents of *lst* with the elements of *coll*

The methods that modify lists are shown in Table B-17.

Table B-17. List-modifying methods

Method	Result
lst.append(*x*)	Adds *x* to the end of *lst*
lst.extend(*x*)	Adds the elements of *x* at the end of *lst*
lst.insert(*i*, *x*)	Inserts *x* before the *i*th element of *lst*
lst.remove(*x*)	Removes the first occurrence of *x* from *lst*; raises an error if *x* is not in *lst*
lst.pop([*i*])	Removes the *i*th element from *lst* and returns it; if *i* is not specified, removes the last element
lst.reverse()	Reverses the list
lst.sort([*reverseflag*] [, *keyfn*])	Sorts the list by comparing elements or, if *keyfn* is included in the call, comparing the results of calling *keyfn* for each element; if *reverse flag* is true, the ordering is reversed; *keyfn* and *reverseflg* must be specified as keyword arguments, not positionally

Mappings

A *mapping* is a *mutable unordered* collection of *key/value pairs*. The only mapping type used in this book is `dict`. Expressions and statements for dictionaries are documented in Table B-18.

Table B-18. Dictionary expressions and statements

Operation	Result
d[*key*]	Returns the value associated with *key*; raises an error if *d* does not contain *key*
d[*key*] = *value*	Associates *value* with *key*, either adding a new key/value pair or, if *key* was already in the dictionary, replacing its value

Operation	Result
d[*key*] ·= *value*	Augmented assignment, with · any of +, -, *, /, //, %, or **; raises an error if *d* does not have a value for *key* or if the value is not numeric
del *d*[*key*]	Deletes *key* from the dictionary; raises an error if *d* does not contain *key*

Table B-19 lists methods that are unique to `dict` objects.

Table B-19. Methods unique to dictionaries

Method	Result
d.get(*key*[, *default*])	Like *d*[*key*], but does not cause an error if *d* does not contain *key*; instead it returns *default*, which, if not provided, is None
d.setdefault(*key*[, *default*])	Like *d*[*key*] if *key* is in *d*; otherwise, adds *key* with a value of *default* to the dictionary and returns *default* (if not specified, *default* is None)
d.pop(*key*[,*default*])	Like del *d*[*key*], but does not cause an error if *d* does not contain *key*; instead it returns *default*, which, if not provided, is None
d1.update(*d2*)	For each key in *d2*, sets *d1*[*key*] to *d2*[*key*], replacing the existing value if there was one
d.keys()	Returns a special sequence-like object containing the dictionary's keys
d.values()	Returns a special sequence-like object containing the dictionary's values
d.items()	Returns a special sequence-like object containing (*key*, *value*) tuples for the dictionary's keys

Streams

A *stream* is a *temporally ordered* sequence of *indefinite length*, usually limited to one type of element. Each stream has two ends: a *source* that provides the elements and a *sink* that absorbs the elements.

File objects

File objects are created by calling `open(`*`filename`*`, `*`mode`*`)`. Tables B-20 and B-21 show the characters that can be included in the *mode* string.

Table B-20. Mode values for opening a file—text or binary

	Mode	Interpretation
t	Text (default)	Characters or strings depending on method
b	Binary	Bytes

Table B-21. Mode values for opening a file—use

	Initial file position	Read?	Write?
`r`	Beginning (default)	Yes	*No*
`w`	Beginning	*No*	Yes
`a`	End	*No*	Yes
`r+`	Beginning	Yes	*No*
`w+`	Beginning	Yes	Yes
`a+`	End	Yes	Yes

Table B-22 describes the methods supported by file objects. Note that the read methods in this table return empty strings when called at the end of the file, and that they may be called repeatedly at the end of the file with no error occurring. Also, although `writelines` is named analogously to `readlines`, it does not add end-of-file characters to the lines it writes; if you want them, you need to include them explicitly in the lines in the sequence. Another way to write to a file is to specify it as the value of the optional `file` argument to the function `print`.

Table B-22. File object methods

Method	Result
`file``.read([`*`count`*`])`	Reads and returns *`count`* bytes or to the end of *`file`*, whichever comes first; if *`count`* is omitted, reads to end of file
`file``.readline([`*`count`*`])`	Reads and returns *`count`* lines or to the end of *`file`*, whichever comes first; if *`count`* is omitted, reads to end of file
`file``.readlines()`	Reads and returns lines until the end of *`file`* is encountered
`file``.write(`*`str`*`)`	Writes *`str`* to *`file`*
`file``.writelines(`*`seq`*`)`	Writes the strings contained in the sequence *`seq`* to *`file`*

Generators

A *generator* is a stream that returns values from a series it computes. The only thing you can do with a generator is call `next` on it to get its next value:

`next(`*`generator`*`[, `*`default`*`])`
: Gets the next value from the *`generator`* object. If *`generator`* has no more values to produce, returns *`default`* if specified in the call; otherwise, raises the `StopIteration` exception.

Generators may be created in one of two ways:

- A function definition that uses `yield` in place of `return` returns a generator object when it is called.

- The value of an expression with syntax like a comprehension, except with parentheses instead of brackets or braces, is a generator.

Iteration Templates

Most iterations conform to a limited set of patterns. The outlines in this section's tables use the following abbreviations:

–item–
: Any use of `item`

?–item–?
: Boolean expression using `item`

–item1, item2–
: Any use of two items together

·
: Any operator (or function call)

fn
: Any function name

Basic Iteration Templates

The iteration templates are outlined in Table B-23.

Table B-23. Basic iteration templates

Section	Action	Code outline
"Do" on page 114	*Do*	**for** item **in** *collection*: *–item–*
"Collect" on page 115	*Collect* (comprehension usually better)	result = [] **for** item **in** *collection*: result.append(*–item–*) **return** result
"Combine" on page 117	*Combine*	result = *initial_value* **for** item **in** *collection*: result = result · *–item–* **return** result
"Combine" on page 117	*Count*	count = 0 **for** item **in** *collection*: count += 1 **return** count
"Combine" on page 117	*Collection Combine*	result = [] **for** item **in** *collection*: result += *fn*(item) **return** result

Filtering Templates

Templates in which operations are applied only to elements meeting criteria expressed in some Boolean test are summarized in Table B-24.

Table B-24. Filtering templates

Section	Action	Code outline

"Search" on page 120	*Search*	`for item in collection:`
		`    if ?–item–?:`
		`        return item`

"Filter" on page 122	*Filtered Do*	`for item in collection:`
		`    if ?–item–?:`
		`        –item–`
		`return result`

"Filter" on page 122	*Filtered Collect*
	
	(conditional comprehension usually better)
		`result = []`
		`for item in collection:`
		`    if ?–item–?:`
		`        result.append(–item–)`
		`return result`

"Filter" on page 122	*Filtered Combine*	`result = initial_value`
		`for item in collection:`
		`    if ?–item–?:`
		`        result = result · –item–`
		`return result`

"Filter" on page 122	*Filtered Count*	`count = 0`
		`    for item in collection:`
		`        if ?–item–?:`
		`            count += 1`
		`return count`

Other Kinds of Templates

A few other templates are described in Table B-25.

Table B-25. Other templates

Section	Action	Code outline

"Nested iterations" on page 126	*Nested Do*
	
	(with similar variations for other kinds of templates)
		`for item1 in collection1:`
		`    for item2 in collection2:`
		`        –item1,item2–`

"Recursive iterations" on page 128	*Recursive Iteration*	A function definition that implements an iteration during which the function calls itself (or calls another function that calls it, etc.); needed for recursive structures

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

Mitchell L Model has worked in a wide range of platforms, languages, technologies, and domains. His specialties include object technology, knowledge representation, user interfaces, distributed computing, and software development practices, and he is an inveterate software tool builder. He has held academic appointments at Brown, Brandeis, and Wesleyan Universities and has worked at a number of early-phase startup companies. As this book went into production, he joined a new synthetic biology startup. For much of his career, he has been an independent consultant, providing training, mentoring, tools, and support to software development groups learning to use new technologies and practices. He has written and taught many professional programming and technology courses and wrote one of the first C++-based data structure books.

Since 1994 Mitchell has been working primarily in bioinformatics, becoming captivated by the complexities of the biological phenomena that it addresses. He was the senior technologist in Millennium Pharmaceutical's large bioinformatics department from 1994 through 2001. In recent years, he has been teaching courses in Northeastern University's Professional Masters in Bioinformatics program.

Mitchell is devoted to teaching, always seeking new ways to effectively communicate the conceptual beauty of computer science. What gives him the greatest satisfaction is coaching students and professionals in the art of software development to enable them to work more effectively and creatively.

Colophon

The animal on the cover of *Bioinformatics Programming Using Python* is a brown rat (*Rattus norvegicus*), also known as a common rat, sewer rat, Norway rat, or wharf rat. One of the largest members of the Muroidea family, the brown rat is 10 to 15 inches long with a 6- to 8-inch long tail. Its fur is coarse and mostly brown or gray. This rat has very sharp hearing and smell, but poor vision. It is omnivorous and nocturnal, and lives almost everywhere humans live, particularly in cities. Like other rodents, brown rats may carry pathogens and spread disease.

Contrary to the brown rat's genus name—*norvegicus*—it did not originate in Norway. It was named by an 18th-century British naturalist who mistakenly believed that the rat had migrated to England on Norwegian ships in the early 1700s. By the end of the 19th century, scientists had established that the brown rat had most likely originated in China. Today the brown rat has spread to all continents and is the dominant rat in Europe and North America—making it the most successful mammal on earth after humans.

The cover image is from the Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.